Academic year: 2013/2014

Trabajo Fin de Grado / Final Bachelor Thesis

Implementing an Embedded Linux System in Xilinx Zynq

Author:
Ginés Hidalgo Martínez

Spanish tutor:
Francisco Javier Toledo Moreo

Austrian tutor:
Franz Fidler

ABSTRACT:

The final achievement of this project is to develop and implement a custom and Embedded Linux Operating System (OS) integrated with a specific PL peripheral. This OS will be developed on ZedBoard (Zynq Evaluation & Development Board) development kit, Xilinx's Zynq-7000 All Programmable System on Chip which contains a dual core ARM Cortex-A9 and a 7 Series FPGA Artix-7.

Therefore, how to create, configure, build and implement an Embedded Linux OS on ZedBoard will be explained in detail during this Final Bachelor Thesis. The entire development process has been structured in several chapters according to the logic order which should be followed to perform it. An overview of the chapters is showed below:

Chapter 1: overall vision of the goals of this project and why perform it.

Chapter 2: short introduction to Embedded Systems, to ZedBoard, to Xilinx Design Environment, and to some GNU tools.

Chapter 3: configuration and implementation of a “Basic” and Custom Embedded Linux OS. Note the importance which BuildRoot will have in this process.

Chapter 4: configuration and implementation of a complete Ubuntu Linux OS. Note that this Ubuntu version can be used as any generic PC.

Chapter 5: summary of the achieved objectives and the respective conclutions.

Chapter 6: the bibliography which has been used to perform this thesis.

Appendix 1: the whole process (i.e. step by step) of developing and implementing the two different operating systems on ZedBoard.

Appendix 2: the tools and programs which have been required before start this thesis.

RESUMEN DEL PROYECTO:

El objetivo final de este proyecto es el desarrollo y la implementación de un Sistema Operativo (SO) Linux Embebido personalizado. Este SO será desarrollado sobre el kit de desarrollo ZedBoard (Zynq Evaluation & Development Board), el cual consiste en un sistema Xilinx's Zynq-7000 All Programmable System on Chip, que se puede dividir en un procesador ARM Cortex-A9 de doble núcleo, y en una 7 Series FPGA Artix-7.

Por lo tanto, cómo crear, configurar, construir e implementar este SO Linux Embebido personalizado sobre la ZedBoard, será profundamente explicado durante todo este Trabajo de Fin de Grado. El proceso de desarrollo ha sido estructurado en 6 capítulos y 2 apéndices, acorde con el orden lógico y temporal que debería ser seguido para implementar este proyecto. A continuación, se muestra un resumen de cada capítulo:

Capítulo 1: visión general de todos los objetivos de este proyecto y por qué son interesantes.

Capítulo 2: breve introducción a los sistemas embebidos, a la placa ZedBoard, al entorno de desarrollo Xilinx Design Environment y a algunas herramientas GNU.

Capítulo 3: configuración e implementación de un básico y personalizado SO Linux Embebido. Destacar la importancia que BuildRoot tendrá en este proceso.

Capítulo 4: configuración e implementación de un completo SO Ubuntu Linux. Destacar que esta versión de Ubuntu puede perfectamente ser utilizada para las mismas tareas que cualquier ordenador convencional.

Capítulo 5: resumen de todos los logros alcanzados y respectivas conclusiones.

Capítulo 6: bibliografía utilizada durante este proyecto.

Apéndice 1: proceso íntegro (paso a paso) sobre cómo desarrollar e implementar los dos SO que han sido mencionados previamente sobre la ZedBoard.

Apéndice 2: programas que deben ser instalados antes de empezar este proyecto.

“Only those who dare to fail greatly can ever achieve greatly.”

Robert Francis Kennedy

Thanks

I would like to specially thank to my UPCT advisor Francisco Javier Toledo Moreo for bringing me the opportunity of working in this project, helping me becoming a more competent person.

I would like also to thank in the same way to my advisor in Austria, Franz Fidler, for also helping me with this project in my Erasmus term.

In addition, I would like to thank to my colleague Sebastián Cánovas Carrasco for sharing his knowledge and helping each other.

Finally, thanks to all the people that loves me and cares about me for their unconditional support in every important moment of my life.

Table of Contents

Table of Figures

Table of Commands

Chapter 1: Introduction

1.1. Overview and Targets

The final achievement of this project is to develop and implement a custom and **Embedded Linux Operating System** (OS). This OS will be developed on ZedBoard (Zynq Evaluation & Development Board) development kit. Therefore, how to create, configure, build and implement an Embedded Linux OS on ZedBoard will be explained in detail during this Final Bachelor Thesis.

Nevertheless, and before starting, it is necessary to know more about the board. The ZedBoard use a **Zynq-7000 SoC architecture**, which includes the ARM Cortex-A9 processing system (PS) and the 7 series programmable logic (PL). The individual components which comprise the PS such as I/O peripherals, clocking, interrupt, AXI interfaces and memory controllers are also detailed briefly. Note the relevant efficient PL-to-PS interfacing including processing interrupts generated from a PL peripheral. Consequently, the ZedBoard is the perfect platform to develop and implement the Embedded Linux OS.

Xilinx and its whole family of programs will be the main tool for achieving this goal. The Xilinx program PlanAhead includes all necessary tools (map, place and route, synthesis, implementation and BitStream generation tools) to build and design any typical process flow for FPGA. In addition to these functionalities, it allows multiple run attempts with different RTL (Register Transfer Level) source versions, constraints or different strategies for synthesis and/or implementation.

Nevertheless, these tools are not enough; some **GNU tools** are also required to get this achievement. There are different GNU Linux versions on the Internet which can be downloaded and implemented on ZedBoard for free. Furthermore, there are a wide variety of tutorials and documentation of each one.

Therefore, it will be explained in detailed two different options and set of techniques to implement the Linux Operating System on ZedBoard:

- Developing a **not-graphical**-user-interface **custom Embedded Linux OS** which will be designed exclusively for the specific purposes (e.g. for the specific and required PL peripheral).
- Implementing a complex HDMI **graphical** user-interface **GNU pre-designed Embedded Linux OS** in which specific PL peripheral can be easily added.

They have **different** but interesting and important **advantages**, as it will be showed in the following chapters. Because of this, both of them will be developed and implemented on ZedBoard before thinking which one will be finally chosen to implement the PL peripheral.

Finally, note that "Chapter 3: Custom Embedded Linux OS on ZedBoard" and "Chapter 4: Ubuntu Linux OS on ZedBoard" are **closely related** to "Appendix 1: Guide – Linux on ZedBoard step by step". These chapters are complementary to this appendix and vice versa; they will deal with the development and implementation of the Linux OS. Nevertheless, they will do it in a different form.

- The **third and fourth chapters** will be focus on the **idea**, on the concepts, on answering the questions "what must be do", "why must be do" and "which ways are there available" to develop and implement the Linux OS.
- The **appendix** will be focused on the exactly **steps** to achieve the target, on answering "how must be exactly done to develop and implement it".

Therefore, these chapters are strongly recommended before performing this appendix and this appendix is strongly recommended after reading these chapters in order to better understand the ideas exposed in the whole thesis.

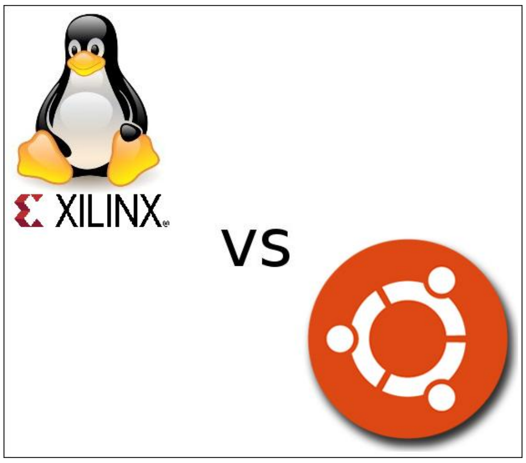


Figure 1: Two Showed Options to Develop and Implement a Linux OS

1.2. Why Develop and Implement an Embedded Linux Operating System

There are a **countless number of reasons** to develop and implement an embedded Linux system. The most relevant will be explained during this section and are summarize below.

- Community support and possibility of taking part into it.
- Devices coverage.
- Eases the new features testing.
- Full control.
- Low cost.
- Platform reusage.
- Quality.
- RTOS (Real Time Operating System) possibility.

After knowing the major advantages of embedded Linux systems, it will be perfectly clear the usefulness of knowing how to develop and implement a system of this kind.

1.2.1. Community Support and Possibility of Taking Part into It

Linux is an open-source code; therefore, there are a great number of developers and user communities sharing their knowledge and code. This results in a high-quality support in which anyone can directly contact with many developers who are working or have been working in the same topic.

In addition, these communities are usually internet communities, allowing 24-hour availability to the user and speeding up the problems resolution.

Finally, there is also the opportunity of taking part into the different communities, for example to bug report; to add new code, versions or patches; etc.

1.2.2. Devices Coverage

Due to the rest of its advantages, there are a great range of systems based on embedded Linux kernel, such as smartphones, tablets, PDAs, smart TVs, machine control, and medical instrument, among others.

1.2.3. Eases the New Features Testing

As already mentioned, Linux is an open-source code. Therefore, it is really easy to get a piece of software and evaluate it. It allows studying several options while making a choice. Furthermore, new possibilities and solutions can be investigated.

As a result, it is too much easier and cheaper than purchasing or using proprietary-product trial versions.

1.2.4. Full Control

The developer can have access to the source code for all components, allowing unlimited changes, modifications, and optimizations without vendor lock-in. Therefore, the developer has full control over the software.

Nevertheless, that is not the case of proprietary embedded operating systems, where the opposite is the case.

1.2.5. Low Cost

Being an open-source includes being free of charge. Therefore, this free software can be duplicated on as many devices as it was necessary with no costs.

It is one of the key advantages, and it can be considered that all other benefits have been produced as a consequence of this advantage.

1.2.6. Platform Re-usage

Linux already provides many components and code for standard well-know functions, such as libraries, multimedia, graphics, protocols, etc.

It allows quickly developing complex products, based on easier, already available components. Therefore, it is not necessary to re-develop the same code by different developers.

Being able to re-use the components and code is another of the key advantages of embedded Linux. It results from the rest of advantages of embedded Linux over proprietary embedded operating systems.

1.2.7. Quality

The open-source components are widely used, in a great multitude of systems. Therefore, a large number of users develop different embedded Linux components and share their knowledge, allowing designing a high quality system with high quality components.

1.2.8. RTOS (Real Time Operating System) Possibility

Another benefit of using an embedded Linux RTOS over a traditional proprietary RTOS is that the Linux community tends to support new IP and other protocols faster than RTOS vendors do.

1.3. Why on ZedBoard

There are a wide range of FPGAs, such as Artix, Kintex, Spartan, Virtex, Zynq ZC70X, etc. Nevertheless, ZedBoard is the chosen board to be used and programmed in this case. Therefore, the **reasons** to select ZedBoard instead of any of the previous boards are showed below and will be briefly explained:

- Different memories types
- Dual core ARM Cortex-A9
- Great variety of peripherals
- SoC architecture

1.3.1. Different Memories Types

ZedBoard includes several kinds of memories, such as a 512MB DDR3 memory, a 256MB flash memory and a SD slot. This gives the flexibility of allowing small-size systems to be stored in the flash memory, with the advantages that this kind of memory involved; and allowing huge-size systems to be stored in an external SD Card.

Therefore, a very fast or a heavy embedded Linux can be developed on ZedBoard.

1.3.2. Dual Core ARM Cortex-A9

ARM is present in most of current Smartphones (about 95% in 2010), and a wide range of smart TVs and laptops (35% and 10% respectively). Therefore, it is the perfect processor for the board.

In addition, ZedBoard not only includes a simple core ARM, but also includes a dual core ARM Cortex-A9. Therefore, the processor will not be a bottleneck for the developed applications on the board in any case.

1.3.3. Great Variety of Peripherals

The ZedBoard provides a wide range of interfaces to connect the most common peripherals and devices, such as monitors, keyboards, speakers, internet connection, etc. The most important interfaces which the ZedBoard provides are the followings:

- Audio line-in, line-out, headphone and microphone.
- Ethernet.
- HDMI and VGA.
- OLED display.
- SD Card.
- USB.

Therefore, the embedded Linux version will be able to use a monitor, a mouse and a keyboard; and will be able also to have internet connection.

1.3.4. SoC Architecture

The ZedBoard is an evaluation and development board based on the Xilinx Zynq-7000 All Programmable SoC (System-on-a-chip). This board allows creating a Linux, Android, Windows or other OS/RTOS-based design.

Therefore, the ZedBoard is not only a FPGA, but it is a Programmable SoC device. But, what is the difference between a FPGA and a programmable SoC device? The main difference is that the SoC combines the processing system (PS) with the programmable logic (PL); as a result, it has a higher speed and a less size than a traditional FPGA.

In particular, the ZedBoard combines a dual Corex-A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells.

Chapter 2: ZedBoard Platform and Xilinx Zynq-7000 SoC

2.1. Embedded Systems

It will be developed and implemented an Embedded Linux Operating System; therefore, before explaining the operation and features of the ZedBoard, it is necessary to introduce and discuss embedded systems.

Embedded systems are computer-based systems designed for specific functions function, usually within a larger mechanical or electrical system, often with real-time computing constraints. They are embedded as part of a complete device often including hardware and mechanical parts. These systems are used rather than general-purpose systems, such as a laptop or a PC, which are designed to be flexible and to cover a wide range of end-user needs.

One of the early first embedded systems was developed by IBM for the Gemini Project, which was found on an on-board computer integrated with other spacecraft systems. Since this embedded system, up to now, there has been a **significant development and evolution** of these devices. They are not always small parts within a larger device which is used like a more general purpose device; many of them consist on standalone devices. Some examples are show below:

- **Standalone** embedded systems:
  - Smartphones.
  - Smart TVs.
  - Mp3.
  - Digital clocks.
- Embedded system **integrated** in more **general purpose devices**:
  - In cars: airbag control unit, closing velocity sensor, fuel injection control, ABS, etc.
  - In factories: temperature control unit, radiation control unit, etc.

Embedded systems rove from not **user-interface** at all to complex graphical user-interfaces depending on the purpose to which are made for. Nevertheless, they usually have at least a basic interface for the developer or for its maintenance, such as a Serial Communication Interface port to communicate with another device. Therefore, in this thesis, two embedded Linux OS will be implemented.

- A **not-graphical**-user-interface basic Linux OS, in which the commands will be introduced through a command window terminal using a Serial Communication Interface port.
- An Ubuntu Desktop Linux OS, provided with a complex HDMI **graphical** user-interface, such as any Ubuntu version installed in a generic PC.

Otherwise, one of the major problems of embedded systems are **system overloads** or bottlenecks. They cause the increment of the data latency, delay interrupt handling, and lower data throughput, among others. The Parallel Processing, which FPGA can achieve, is efficient for critical system performance but a central controller and memory management is needed. Therefore, one traditional solution is building a discrete hybrid system, using a microcontroller together with a FPGA, which provides the best of both of them. Nevertheless, there are some **disadvantages**, among others:

- Limited bandwidth between the two integrated circuits.
- Increasing power consumption.
- Increasing size and complexity.

Therefore, it is required a better solution to this problem. The current technology needs the **union** between the **Processing System (PS) and** the **Programmable Logic (PL)** in a single device. In this context, the Zynq-7000 All Programmable System on Chip (SoC) appears.

A System on a Chip or **System on Chip (SoC)** is an integrated circuit (IC) which integrates all components of a computer or other electronic system into a single chip, such as digital signal, analog signal, mixed-signal, radio-frequency functions, etc.

The contrast with a **microcontroller** is one of degree. Microcontrollers typically have a few RAM and they are often really single-chip-systems, whereas the term SoC is used for more powerful processors, capable of running heavy software such as a desktop version of Windows or Linux, using external memory chips (flash and/or RAM) and disposing of several external peripherals.

2.2. Xilinx Zynq-7000 All Programmable SoC Architecture

Based on the Xilinx All Programmable SoC (AP SoC) architecture, the Zynq-7000 All Programmable System on Chip (SoC) System on Chip is not a simple FPGA; it is a 28nm **programmable-logic** fabric 7 Series family FPGA **coupled** with a dual ARM Cortex-A9 MP Core **processor** in a single chip, allowing a wide range of specific interface functions, such as gigabit transceivers, high performance I/Os, high throughput AXI (Advanced eXtensively Interface), thousands PS to PL connections, among others. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces and a rich set of I/O peripherals.

This two-chip combo All Programmable SoC causes a **lower cost, complexity, size and power consumption** of the system. At the same time, the system **performance** is **increased**. Therefore, this tight integration between the ARM-based PS and the on-chip PL creates unlimited possibilities for designers to add virtually any peripheral or create custom accelerators which can extend the system performance.

The range of devices in the **Zynq-7000 AP SoC family** enables designers to target cost-sensitive as well as high-performance applications from a single platform using industry-standard tools. Additionally, each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between them.

The **PS** and the **PL** are on **separate power domains**, allowing the possibility of powering down the PL if required. Additionally, the processors in the PS always boot first. It is also interesting to note that the PL can be configured as part of the boot process or later at some point in the future. In addition, it can be completely reconfigured or used with partial, dynamic reconfiguration (PR). Moreover, PR allows configuration of a portion of the PL. The functional blocks of the Zynq-7000 AP SoC are showed below.

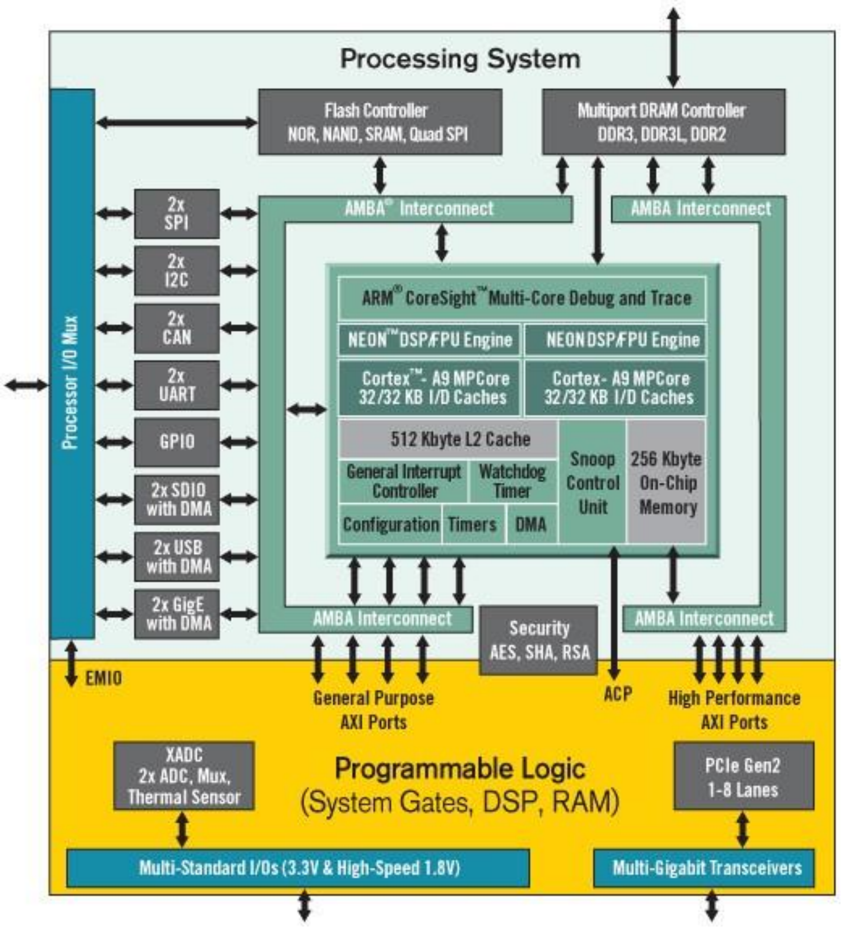

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Snoop Control Unit
256 Kbyte On-Chip Memory
Configuration
Timers
DMA
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 2: Zynq-7000 Diagram

2.3. ZedBoard Platform

2.3.1. ZedBoard Features

ZedBoard (Zynq Evaluation & Development Board) is a single-board computer based on Xilinx's Zynq device family. It uses a Xilinx Zynq Z-7020 Zynq device (dual core ARM Cortex-A9 cores ~800MHz paired with a Xilinx Artix 7 FPGA).

ZedBoard is intended to be a community development platform evaluation and development board based on the above-mentioned Xilinx Zynq-7000 All Programmable System on Chip. Combining the dual Cortex-A9 Processing System with an 85000 7-Series Programmable Logic cells, the board contains interfaces and supporting functions to enable a wide range of applications. The ZedBoard features are summarized below:

- **Processor**: Zynq-7000 AP SoC XC7Z020-CLG484-1.
    - Up to 667MHz operation.
    - NEON Processing/FPU Engines.
    - Dual core.

- **Memories**:
    - 512MB DDR3 memory.
    - 256Mb Quad SPI Flash.
    - 4GB SD Card.
- **Communication**:
    - Onboard USB-JTAG programming.
    - 10/100/1000 Ethernet.
    - USB OTG 2.0 and USB-UART bridge.
- **Clocking**:
    - 33.33333MHz clock source for PS.
    - 100MHz oscillator for PL.
- **Display**:
    - HDMI output supporting 1080p60 with 16-bit resolution color.
    - VGA output with 12-bit resolution color.
    - 128x32 OLED display.
- **Audio**:
    - AudiLine-in.
    - Line-out.
    - Headphone.
    - Microphone.
- **General Purpose I/O**:
    - 9 user LEDs (1 PS, 8 PL).
    - 7 push buttons (2 PS, 5 PL).
    - 8 switches (PL).
- **Configuration and Debug**:
    - Onboard USB-JTAG interface.
    - Xilinx Platform Cable JTAG connector.
- **Connectivity**:
    - USB-JTAG Programming.
    - 5 Digilent Pmod headers.
    - FMC (Low Pin Count) connector.
    - USB OTG 2.0 (Device/Host/OTG).
    - TwReset Buttons (1 PS, 1 PL).
    - ARM Debug Access Port (DAP).
    - Xilinx XADC Header.

2.3.2. ZedBoard Hardware Block Diagram

The next figure shows the ZedBoard Hardware block diagram:

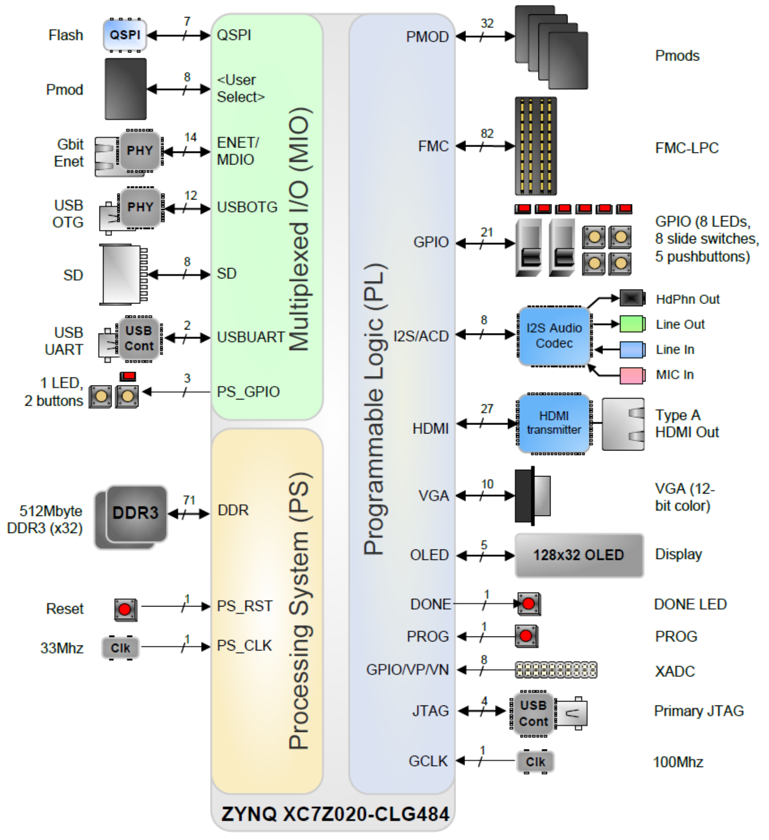


Figure 3: ZedBoard Block Diagram

2.3.3. Zynq Bank Pin Assignments

The following figure shows the Zynq bank pin assignments on ZedBoard:

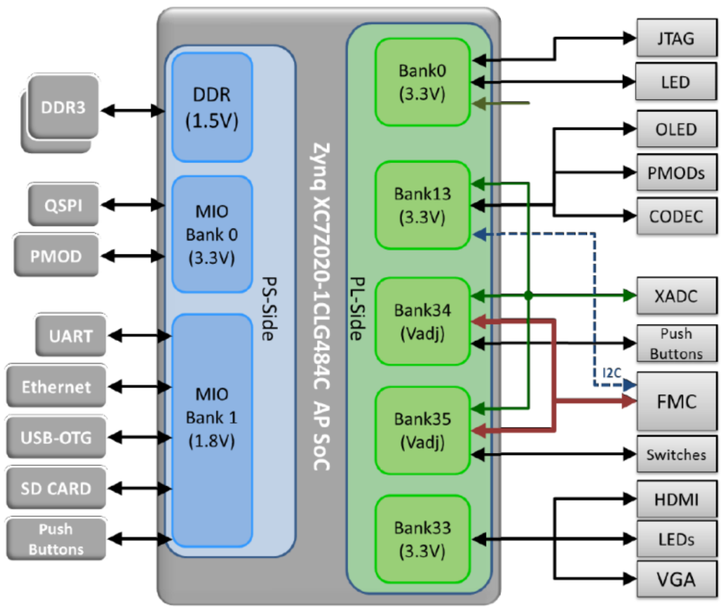


Figure 4: Zynq Z7020 CLG484 Bank Assignments

2.4. Required Software

Up to this moment, the thesis targets and the required hardware to achieve them have been known. Nevertheless, it is also necessary briefly explain the desired tools to implement and develop this software.

2.4.1. ISE Design Suite (Xilinx Design Environment)

Xilinx Design Environment will be the main tool to achieve this goal. Xilinx is an American technology company, primarily a supplier of programmable logic devices. It is known for inventing the field programmable gate array (FPGA) and as the first semiconductor company with a fabless manufacturing model (contracts out their production rather than owning its own factory). It was found in Silicon Valley in 1984.

Xilinx is the world's leading provider of All Programmable FPGAs, SoCs and 3D ICs. These industry-leading devices are coupled with a next-generation design environment and IP to serve a broad range of customer needs, from programmable logic to programmable systems integration. Actually, the Zynq-700 AP SoC is a Xilinx product.

Therefore, the **ISE Design Suite** designed by Xilinx is the desired and ideal development environment for any Zynq-700 AP SoC device, such as ZedBoard. ISE Design Suite is a proven and mature development environment for All Programmable devices. ISE includes the PlanAhead Design and Analysis tools for the ZedBoard. Its embedded processing component includes PlanAhead, Xilinx Platform Studio (XPS) and the Software Development Kit (SDK).

The Zynq **Processing System (PS)** may be used **without** anything programmed in the **Programmable Logic (PL)**. However, in order to use any soft IP in the PL, or to route PS dedicated peripherals to device pins for the PL, programming of the PL is required. A Zynq PS-only project can be completed in XPS and SDK standalone.

Nevertheless, once any piece of PL logic with the PS is required, a greater range and power tool like PlanAhead is required. **PlanAhead** provides a central cockpit for design entry in RTL, synthesis, verification and BitStream generation. PlanAhead offers integration with XPS for embedded processor design (including access to the Xilinx IP catalog), and SDK to complete the embedded processor software design. It includes all necessary tools (map, place and route, synthesis, implementation and BitStream generation tools) to build and design any typical process flow for FPGA. In addition to these functionalities, it allows multiple run attempts with different RTL (Register Transfer Level) source versions, constraints or different strategies for synthesis and/or implementation.

Once the hardware project has been created with PlanAhead, using the XPS help, it will be exported to SDK in order to develop the required software.

In conclusion, the **Xilinx programs** which will be **used** during this thesis are:

- PlanAhead.
- Xilinx Platform Studio (XPS).
- Software Development Kit (SDK).

2.4.2. GNU Tools

ISE Design Suite will be enough if a Standalone system is developed and implemented. Nevertheless, in this case, it will be developed an embedded Linux OS; therefore, these tools are not enough. Some **GNU tools** are also required to get this achievement and to build the specific Linux required files. There are different GNU Linux versions on the Internet which can be downloaded and implemented on ZedBoard with no cost. Furthermore, there are also available tools which support the Linux-OS development by selecting the necessary packages to configure it and create the desired Root File System Image.

The main used tools are showed below and briefly explained during this section.

- U-Boot.
- Linux kernel.
- BuildRoot.

2.4.2.1. U-Boot

Das U-boot is a GNU Universal Boot Loader which is frequently used in embedded Linux devices. It was developed by Magnus Damm like a bootloader for PC, in 1999 for the Wolfgang Denk Company. The current name “Das U-Boot” adds a German definite article, to create a bilingual pun on the German word for “submarine”.

Das U-Boot is the richest, most flexible and most actively developed open-source Boot Loader available for Embedded Linux OS. It is available for a number of **different** computer **architectures**, not only the ARM architecture of the ZedBoard, but also others such as 68k, AVR32, Blackfin, MicroBlaze, MIPS, Nios, PPC, and x86, among others. The development of U-Boot is closely related to Linux: some parts of the source code originate in the Linux source tree, and there are some header files in common.

Xilinx provides an official Xilinx U-Boot repository, **u-boot-xlnx**, which includes U-Boot to run on Xilinx boards. It is based on the source code from the DENX Software Engineering Git tree repository.

The most common **U-Boot starting process** is shown below:

- U-boot is loaded with the desired parameters.
- Environment variables which contain U-boot parameters are loaded.
- U-boot loads the embedded-Linux-OS kernel.

2.4.2.2. Linux Kernel

It is required a **Linux Kernel** repository which is able to build the required Linux files, such as the Device Tree Binary file, the Linux Kernel file and the Root File System Image. The Linux Kernel is an open-source Unix-like operating system kernel used by a variety of operating systems based on it, which are usually in the form of Linux distributions. The Linux kernel is a prominent example of free and open source software.

The Linux kernel was initially conceived and created in 1991 by the Finnish computer science student Linus Torvalds; and rapidly accumulated a countless number of developers and users who adapted code from other free software projects. Therefore, it is developed by contributors worldwide, allowing being constantly updated to the current technology.

In this case, **linux-xlnx** will be used in the third chapter to implement the "basic" Linux OS, which is the official Linux kernel from Xilinx. It is a Linux Kernel repository which provides the Linux Kernel for the whole Xilinx devices family, like Zynq.

In addition, another Linux kernel repository will be used in the fourth chapter to implement the kernel and device tree binary for the Ubuntu Linux OS, the Linux kernel repository of **Analog Devices Inc.**, which includes HDMI interface configuration, among other peripheral configurations.

2.4.2.3. BuildRoot

Building a custom **Embedded Linux OS** involves creating a kernel image with its required libraries, the packets to support the hardware devices and software applications, the dependencies among packages, the required Boot Loader with its required libraries, choose which packages versions and tools are compatibles, build the root file system and the software applications, and an extremely **large** etcetera of other **required tasks**. Therefore, building all without help will waste a lot of time.

For all these reasons, a simple, efficient and easy-to-use **tool to generate embedded Linux** OS through cross-compilation tool is desired. It will simplify and automate the process of building a complete Linux system for an embedded system.

In order to achieve this, **BuildRoot** will be the preferred tool. It is a bunch of Makefiles and patches designed to build a complete embedded Linux distribution, automating the embedded system building process. Its **key features** and functions are summarized below.

- It can build all the required components for the embedded system, cross-compiling toolchain, root filesystem generation, kernel image compilation, Device Tree Blob compilation, and boot-loader (U-Boot) compilation.
- It allows configuration using configuration interfaces, such as the "xconfig" configuration interface.
- It offers a great variety of supplementary software, such as packages, applications and libraries, which can be added to the Embedded Linux.
- It supports multiple filesystem types for the root filesystem image.
- It supports numerous processors and their variants, such as ARM.
- It includes default configurations for several boards, such as ZedBoard.
- It can generate an uClibc cross-compilation toolchain.
- It is a simple structure relying on Makefile language.
- It can configure any combination of these options, independently of each other (e.g. it can be used to build only the Linux Kernel whether it was the only required file).
- If it requires changing the Linux kernel version, it will only require change the download git repository in the configuration file, while the rest of the configuration will remain unchanged. This allows re-using the configuration between different Linux versions.

Therefore, BuildRoot is the ideal tool to build the "Basic" Linux OS, which will be realize in the third chapter.

Chapter 3: Custom Embedded Linux OS on ZedBoard

3.1. Overview

The aim is to build a **custom Embedded Linux OS** on ZedBoard. Xilinx and its whole family of programs will be the main tool for achieving this goal, as already mentioned.

As already mentioned in chapter 2, the chosen Linux OS which will be developed is **Linux-xlnx**, and the main tool which will be used to configure and build it is BuildRoot. It is well worth remembering that the Linux OS is available in its git repository with a complete and detailed documentation: https://github.com/Xilinx/linux-xlnx; the same as BuildRoot is also available in its own git repository: http://git.buildroot.net/buildroot.

BuildRoot will be the main GNU tool used to download, configure and build this Linux OS. Note that this tool can be used to configure and implement most Linux versions available on the Internet in almost any board or FPGA. This is why is very interesting to know how this tool works. It is also possible to develop the OS without BuildRoot, but it would require much more time because of the different advantages of BuildRoot:

- BuildRoot allows not only configure the Linux Kernel and Root File System Image, but also it is able to configure and build also the U-Boot file and the Device Tree Binary.
- It also offers a greater variety of supplementary configurations or programs which can be added to the Embedded Linux.
- If it requires changing the Linux version (Linux-xlnx), it will only require change the download git repository. In addition, the rest of the configuration will remain unchanged. This allows using the same configuration between different Linux versions.

These are some of the main reasons why this tool will be used to develop Linux.

If the documentation of the Linux-xlnx is read, it can be seen that four **files** are **required** in the SD Card to start this Linux OS on ZedBoard:

1. The **Boot Image** ("boot.bin"), which is a binary composite image, consisted on different files. The most simple Linux systems require only three components within the boot image:
   a. The **FSBL** (First Stage Boot Loader).
   b. The **Programmable Logic Hardware BitStream** (optional).
   c. The **U-Boot** (Second Stage Boot Loader).
2. The **Device Tree Binary File** ("devicetree.dtb"), which is obtained from the Device Tree Source File and loaded into the DDR memory by U-Boot. The kernel has to know every detail about the hardware it is working on. For this purpose, it uses the data structure known as Device Tree Blob or Device Tree Binary to describe the hardware.
3. The **Linux Kernel File** ("uImage"), which is also loaded into the DDR memory by U-Boot. It initializes the system hardware and mounts the root file system.

4. The **Root File System Image** ("uramdisk.image.gz"), which is also loaded into the DDR memory by U-Boot. It contains the Operating System itself.

The next figures summarize these steps. On one hand, the first image shows the required files in the SD Card. On the other hand, the second one shows the typical Linux Boot sequence.

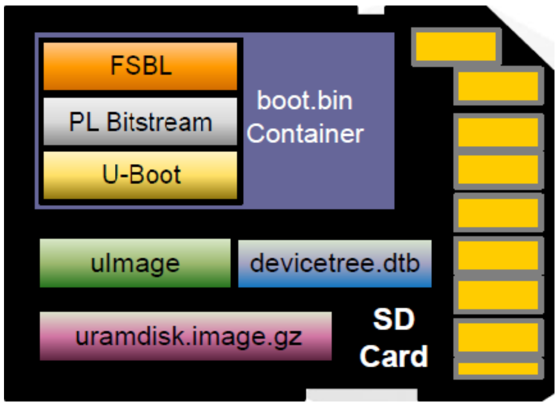


Figure 5: Files Contained in the SD Card

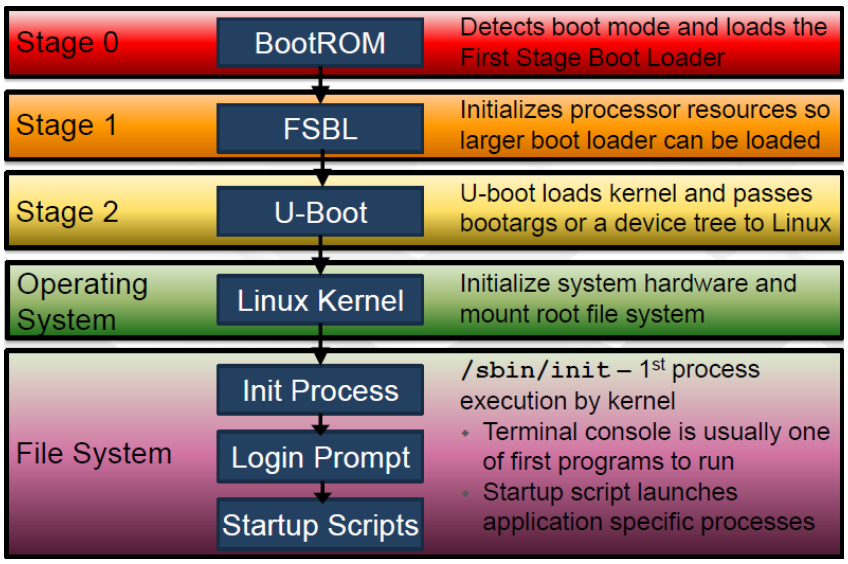

Stage 0
BootROM
Detects boot mode and loads the First Stage Boot Loader
Stage 1
FSBL
Initializes processor resources so larger boot loader can be loaded
Stage 2
U-Boot
U-boot loads kernel and passes bootargs or a device tree to Linux
Operating System
Linux Kernel
Initialize system hardware and mount root file system
File System
Init Process
Login Prompt
Startup Scripts
/sbin/init – 1st process execution by kernel
Terminal console is usually one of first programs to run
Startup script launches application specific processes

Figure 6: Typical Linux Boot Sequence Overview

All these files will be explained in further detail in this chapter. Despite of the **Linux Boot Sequence**, the order in which these files will be made is different. This is because BuildRoot will build the U-Boot file, the Device Tree Binary file, the Linux Kernel file and the Root File System Image at the same time. Nevertheless, the whole "boot.bin container" cannot be built until the U-Boot file will be made. Moreover, the BitStream is the first required file which is created.

Nevertheless, before performing this task, a **new project** must be created and configured for ZedBoard in PlanAhead, in order to be able to create a specific FSBL and BitStream files for the specific hardware, the ZedBoard.

Therefore, the programs and steps to create, configure, build and implement a “Basic” Embedded Linux Operating System with BuildRoot on ZedBoard are summarized below.

1. **PlanAhead:**
   a. Creating a PlanAhead Project.
   b. Adding an embedded source.
2. **Xilinx Platform Studio (XPS):**
   a. Configuring the Hardware Platform (e.g. peripherals, clocks, DDR3 memory, etc).
3. **PlanAhead:**
   a. Design Constraints.
   b. Top HDL Module.
   c. Hardware Platform Building.
4. **Software Development Kit (SDK):**
   a. Creating the FSBL (First Stage Boot Loader).
5. **BuildRoot:**
   a. Configuring the OS.
   b. Building the custom OS.
   c. Obtaining the U-Boot file, the Device Tree Binary file, the Kernel Image and the Root File System Image.
6. **Software Development Kit (SDK):**
   a. Creating the boot image (“boot.bin”) from the FSBL, the BitStream file and the U-Boot file.
7. **SD Card:**
   a. Copying the four files in the SD Card.
8. **Tera Term:**
   a. Running Linux on ZedBoard.

Finally, remember that this chapter is **closely related** to “Appendix 1: Guide – Linux on ZedBoard step by step”; they both are complementary to each other. This section will be focus on the idea, on the concepts, on answering the questions “what must be do”, “why must be do” and “which ways are there available” to develop and implement the Linux OS. Nevertheless, the appendix will be focused on the exactly steps to achieve the target, on answering “how must be exactly done to develop and implement it”.

Therefore, this appendix is strongly recommended after reading this chapter in order to better understand the ideas exposed in this chapter.

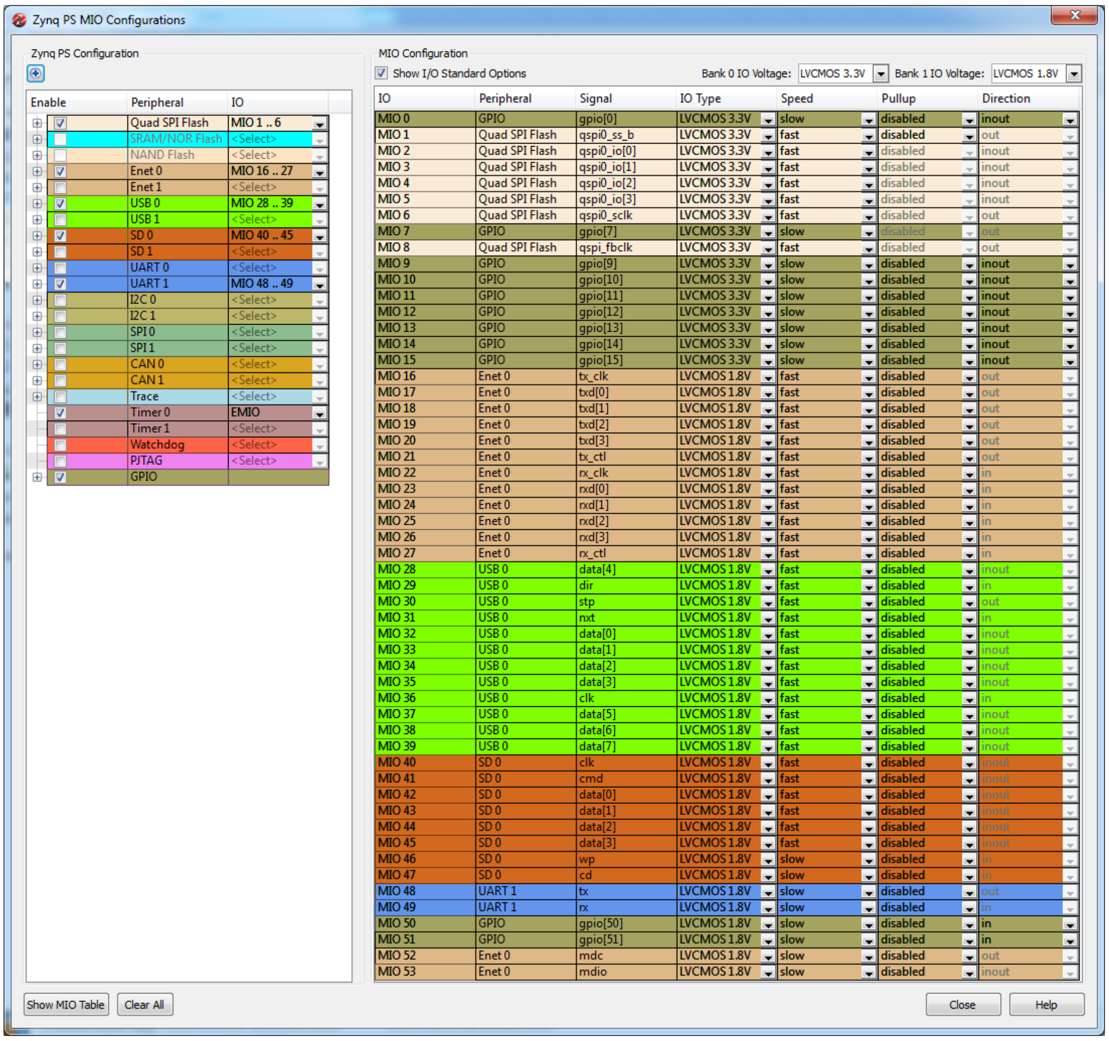
3.2. Creating a Project and Adding Embedded Sources

The first required action will be to create a **Register Transfer Level (RTL) project** to manage the entire System on Chip design flow, where the specific target device, the ZedBoard, must be also selected (Zynq-7000 Family, Package dg484, Speed grade -1 and Temp grade C). A RTL is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.

Hereafter, an **Embedded Source**, the ARM processing system, has to be added and configured. After adding an Embedded Source in PlanAhead, the Xilinx Platform Studio (XPS) will be automatically launched to configure it. The configuration of the peripherals could be made one by one or could be imported from a configuration file available on ZedBoard.org webpage. It will be easier and faster to import it.

After that, many **peripherals** are enabled in the Processing System with some MIO pins assigned to them in coordination with the ZedBoard layout. For example, UART1 is enabled and UART0 is disabled because UART1 is connected to the USB-UART bridge chip on this board.

Another important detail to highlight is that the **peripherals** are not listed in alphabetical order, despite the MIOs numbers are listed in number order. The peripherals are listed from top to bottom **in order of priority** based on either their importance in the system (like the Flash memory) or how limited they are in their possible MIO mappings. The least flexible is at the top (the Flash memory), while the most flexible (GPIO) is at the bottom.



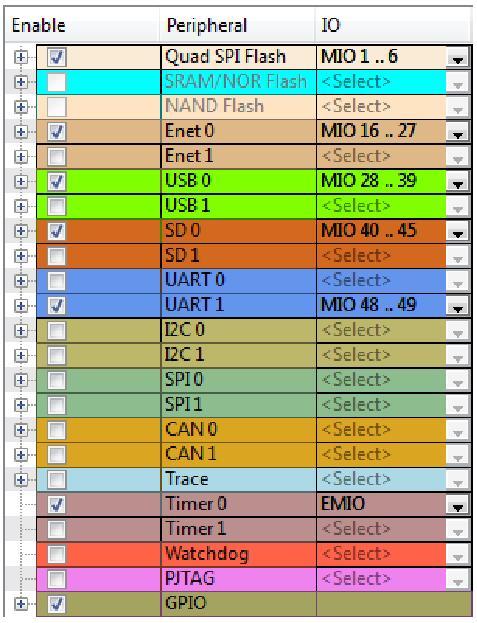
☑ Show I/O Standard Options — Bank 0 IO Voltage: LVCMOS 3.3V — Bank 1 IO Voltage: LVCMOS 1.8V

Enable	Peripheral	IO
✓	Quad SPI Flash	MIO 1 .. 6
☐	SRAM/NOR Flash	<Select>
☐	NAND Flash	<Select>
✓	Enet 0	MIO 16 .. 27
☐	Enet 1	<Select>
✓	USB 0	MIO 28 .. 39
☐	USB 1	<Select>
✓	SD 0	MIO 40 .. 45
☐	SD 1	<Select>
☐	UART 0	<Select>
✓	UART 1	MIO 48 .. 49
☐	I2C 0	<Select>
☐	I2C 1	<Select>
☐	SPI 0	<Select>
☐	SPI 1	<Select>
☐	CAN 0	<Select>
☐	CAN 1	<Select>
☐	Trace	<Select>
✓	Timer 0	EMIO
☐	Timer 1	<Select>
☐	Watchdog	<Select>
☐	PJTAG	<Select>
✓	GPIO	

IO	Peripheral	Signal	IO Type	Speed	Pullup	Direction
MIO 0	GPIO	gpio[0]	LVCMOS 3.3V	slow	disabled	inout
MIO 1	Quad SPI Flash	qspi0_ss_b	LVCMOS 3.3V	fast	disabled	out
MIO 2	Quad SPI Flash	qspi0_io[0]	LVCMOS 3.3V	fast	disabled	inout
MIO 3	Quad SPI Flash	qspi0_io[1]	LVCMOS 3.3V	fast	disabled	inout
MIO 4	Quad SPI Flash	qspi0_io[2]	LVCMOS 3.3V	fast	disabled	inout
MIO 5	Quad SPI Flash	qspi0_io[3]	LVCMOS 3.3V	fast	disabled	inout
MIO 6	Quad SPI Flash	qspi0_sclk	LVCMOS 3.3V	fast	disabled	out
MIO 7	GPIO	gpio[7]	LVCMOS 3.3V	slow	disabled	out
MIO 8	Quad SPI Flash	qspi_fbclk	LVCMOS 3.3V	fast	disabled	out
MIO 9	GPIO	gpio[9]	LVCMOS 3.3V	slow	disabled	inout
MIO 10	GPIO	gpio[10]	LVCMOS 3.3V	slow	disabled	inout
MIO 11	GPIO	gpio[11]	LVCMOS 3.3V	slow	disabled	inout
MIO 12	GPIO	gpio[12]	LVCMOS 3.3V	slow	disabled	inout
MIO 13	GPIO	gpio[13]	LVCMOS 3.3V	slow	disabled	inout
MIO 14	GPIO	gpio[14]	LVCMOS 3.3V	slow	disabled	inout
MIO 15	GPIO	gpio[15]	LVCMOS 3.3V	slow	disabled	inout
MIO 16	Enet 0	tx_clk	LVCMOS 1.8V	fast	disabled	out
MIO 17	Enet 0	txd[0]	LVCMOS 1.8V	fast	disabled	out
MIO 18	Enet 0	txd[1]	LVCMOS 1.8V	fast	disabled	out
MIO 19	Enet 0	txd[2]	LVCMOS 1.8V	fast	disabled	out
MIO 20	Enet 0	txd[3]	LVCMOS 1.8V	fast	disabled	out
MIO 21	Enet 0	tx_ctl	LVCMOS 1.8V	fast	disabled	out
MIO 22	Enet 0	rx_clk	LVCMOS 1.8V	fast	disabled	in
MIO 23	Enet 0	rxd[0]	LVCMOS 1.8V	fast	disabled	in
MIO 24	Enet 0	rxd[1]	LVCMOS 1.8V	fast	disabled	in
MIO 25	Enet 0	rxd[2]	LVCMOS 1.8V	fast	disabled	in
MIO 26	Enet 0	rxd[3]	LVCMOS 1.8V	fast	disabled	in
MIO 27	Enet 0	rx_ctl	LVCMOS 1.8V	fast	disabled	in
MIO 28	USB 0	data[4]	LVCMOS 1.8V	fast	disabled	inout
MIO 29	USB 0	dir	LVCMOS 1.8V	fast	disabled	in
MIO 30	USB 0	stp	LVCMOS 1.8V	fast	disabled	out
MIO 31	USB 0	nxt	LVCMOS 1.8V	fast	disabled	in
MIO 32	USB 0	data[0]	LVCMOS 1.8V	fast	disabled	inout
MIO 33	USB 0	data[1]	LVCMOS 1.8V	fast	disabled	inout
MIO 34	USB 0	data[2]	LVCMOS 1.8V	fast	disabled	inout
MIO 35	USB 0	data[3]	LVCMOS 1.8V	fast	disabled	inout
MIO 36	USB 0	clk	LVCMOS 1.8V	fast	disabled	in
MIO 37	USB 0	data[5]	LVCMOS 1.8V	fast	disabled	inout
MIO 38	USB 0	data[6]	LVCMOS 1.8V	fast	disabled	inout
MIO 39	USB 0	data[7]	LVCMOS 1.8V	fast	disabled	inout
MIO 40	SD 0	clk	LVCMOS 1.8V	fast	disabled	inout
MIO 41	SD 0	cmd	LVCMOS 1.8V	fast	disabled	inout
MIO 42	SD 0	data[0]	LVCMOS 1.8V	fast	disabled	inout
MIO 43	SD 0	data[1]	LVCMOS 1.8V	fast	disabled	inout
MIO 44	SD 0	data[2]	LVCMOS 1.8V	fast	disabled	inout
MIO 45	SD 0	data[3]	LVCMOS 1.8V	fast	disabled	inout
MIO 46	SD 0	wp	LVCMOS 1.8V	slow	disabled	in
MIO 47	SD 0	cd	LVCMOS 1.8V	slow	disabled	in
MIO 48	UART 1	tx	LVCMOS 1.8V	slow	disabled	out
MIO 49	UART 1	rx	LVCMOS 1.8V	slow	disabled	in
MIO 50	GPIO	gpio[50]	LVCMOS 1.8V	slow	disabled	in
MIO 51	GPIO	gpio[51]	LVCMOS 1.8V	slow	disabled	in
MIO 52	Enet 0	mdc	LVCMOS 1.8V	slow	disabled	out
MIO 53	Enet 0	mdio	LVCMOS 1.8V	slow	disabled	inout

Figure 7: Peripheral and MIO Order on ZedBoard

The **boot device** is one of the most important parts. Zynq-7000 allows selecting just one: QSPI, NOR or NAND. SD Card is also a boot option and is show up lower in the list. Only one of the three flash memory types can be selected because the three interfaces are mutually exclusive. For example, if the Quad SPI Flash is selected, NOR and NAND peripherals are grayed out because of that.



Enable	Peripheral	IO
✓	Quad SPI Flash	MIO 1 .. 6
☐	SRAM/NOR Flash	<Select>
☐	NAND Flash	<Select>
✓	Enet 0	MIO 16 .. 27
☐	Enet 1	<Select>
✓	USB 0	MIO 28 .. 39
☐	USB 1	<Select>
✓	SD 0	MIO 40 .. 45
☐	SD 1	<Select>
☐	UART 0	<Select>
✓	UART 1	MIO 48 .. 49
☐	I2C 0	<Select>
☐	I2C 1	<Select>
☐	SPI 0	<Select>
☐	SPI 1	<Select>
☐	CAN 0	<Select>
☐	CAN 1	<Select>
☐	Trace	<Select>
✓	Timer 0	EMIO
☐	Timer 1	<Select>
☐	Watchdog	<Select>
☐	PJTAG	<Select>
✓	GPIO	

Figure 8: Flash Memory Selection

3.2.1. PS PLL Clocks

As already mentioned in the previous chapter, the Zynq-7000 AP SoC's PS subsystem uses a dedicated 33.3333MHz clock source, IC18, Fox 767-33.333333-12, with series termination. The PS infrastructure can generate up to four PLL-based clocks for the PL system. An on-board 100MHz oscillator, IC17, Fox 767-100-136, supplies the PL subsystem clock input on bank 13, pin Y9.

In addition, each PLL must be set to operate in a **specific frequency range**, as given by the datasheet. This range is from 780MHz to 1600MHz for the ZedBoard (-1 device). Three clocks can be chosen as "Clock Source" for the rest of clocks:

- The ARM PLL.
- The DDR PLL.
- The I/O PLL.

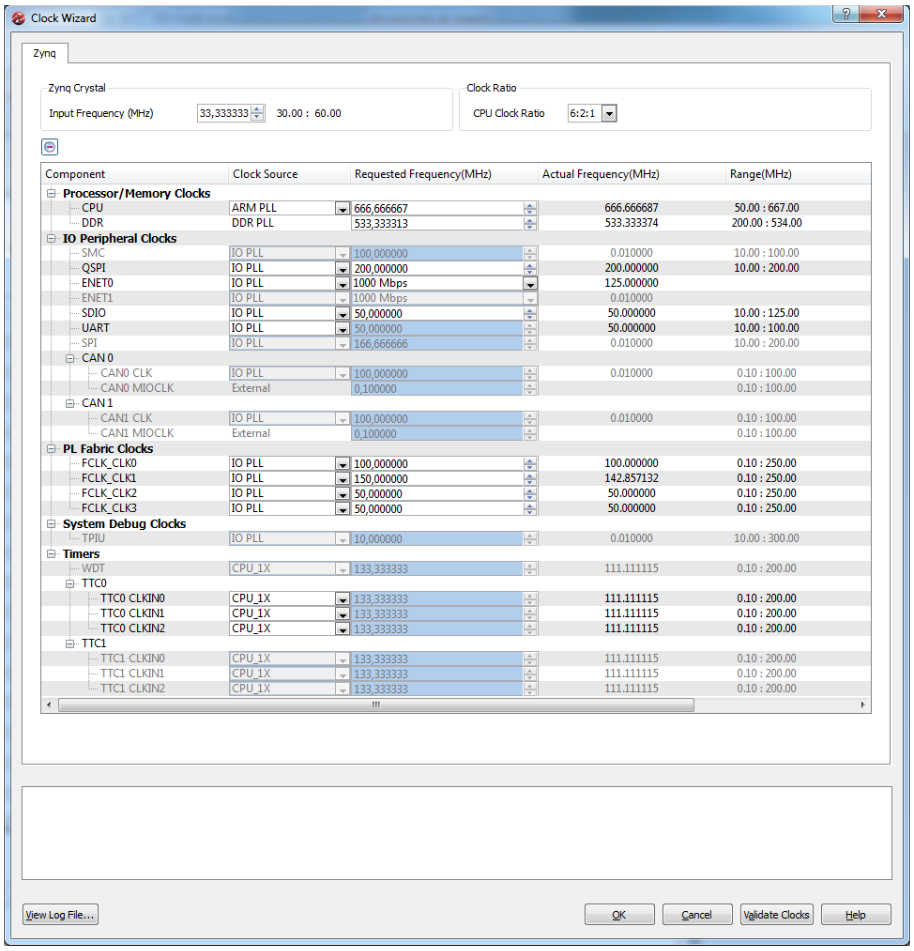
The CPU (ARM) and the DDR frequencies must be **multiples of 33.3333MHz**. On one hand, the ARM PLL is 666,6667MHz which is 33.333MHz · 20. On the other hand, the CPU PLL is 533,3333MHz which is 33.333MHz · 16.

Therefore, **every component** which uses these PLLs must be an **integer** divider of these frequencies. If the frequency selected for a clock is different to this multiplication or division, the program will adjust its clock frequency as close as possible to the requested value.

For instance, the required frequency of the QSPI peripheral interface is 200MHz, if the ARM PLL was be chosen as clock source, the more similar value of frequency would be 190.476MHz (1333.333MHz/7). Nevertheless, if the IO PLL clock is chosen, the exactly frequency would be achieved (33.3333MHz · 6).

On the other hand, despite the PL Fabric Clock "FCLK_CLK1" is set to 150MHz, the actual frequency is 142.857132MHz. That is because 150MHz cannot be exactly achieved with the available dividers and multipliers. The more similar value of frequency is 142.857132MHz (1333.333MHz·3/28).

Finally, note that any value from 30 to 60MHz is accepted as "**Input Frecuency**". However, 33.3333MHz is the standard value of the Fox clock which provides the best performance:



Zynq Crystal — Input Frequency (MHz): 33,333333 30.00 : 60.00

Clock Ratio — CPU Clock Ratio: 6:2:1

Component	Clock Source	Requested Frequency(MHz)	Actual Frequency(MHz)	Range(MHz)
Processor/Memory Clocks				
CPU	ARM PLL	666,666667	666.666687	50.00 : 667.00
DDR	DDR PLL	533,333313	533.333374	200.00 : 534.00
IO Peripheral Clocks				
SMC	IO PLL	100,000000	0.010000	10.00 : 100.00
QSPI	IO PLL	200,000000	200.000000	10.00 : 200.00
ENET0	IO PLL	1000 Mbps	125.000000	
ENET1	IO PLL	1000 Mbps	0.010000	
SDIO	IO PLL	50,000000	50.000000	10.00 : 125.00
UART	IO PLL	50,000000	50.000000	10.00 : 100.00
SPI	IO PLL	166,666666	0.010000	10.00 : 200.00
CAN 0				
CAN0 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN0 MIOCLK	External	0,100000		0.10 : 100.00
CAN 1				
CAN1 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN1 MIOCLK	External	0,100000		0.10 : 100.00
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100,000000	100.000000	0.10 : 250.00
FCLK_CLK1	IO PLL	150,000000	142.857132	0.10 : 250.00
FCLK_CLK2	IO PLL	50,000000	50.000000	0.10 : 250.00
FCLK_CLK3	IO PLL	50,000000	50.000000	0.10 : 250.00
System Debug Clocks				
TPIU	IO PLL	10,000000	0.010000	10.00 : 300.00
Timers				
WDT	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0				
TTC0 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1				
TTC1 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00

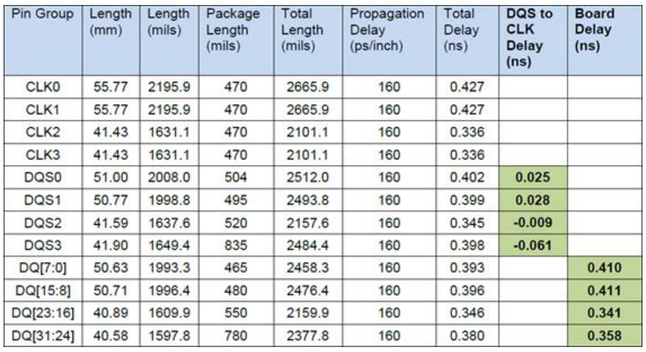
Figure 9: Clock Wizard in XPS after Importing the Configuration File

3.2.2. DDR3 Memory

ZedBoard includes two 32-bit DDR3 memories, totaling 512MB. The PS incorporates both the DDR controller and the associated Physical Layer Interface, including its own set of dedicated I/Os. The DDR3 memory interface speeds up to 533MHz.

The PS7 DDR Configuration screen allows for configuration of the DDR Controller, the Memory Part, and the board details used for DDR interface.

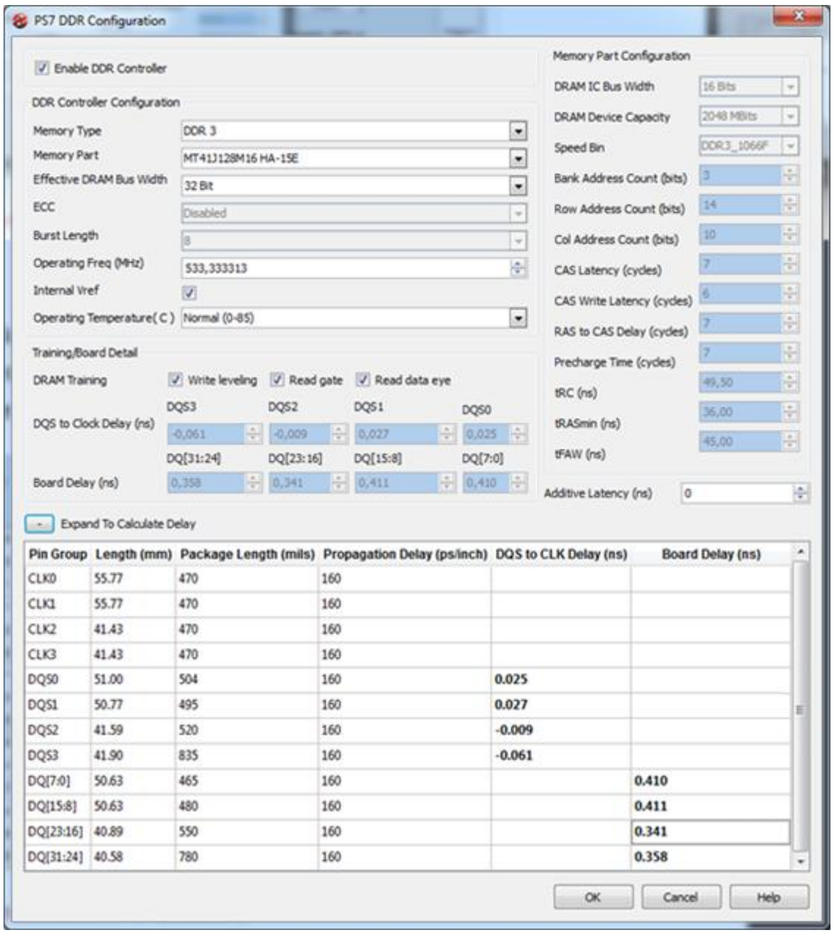
It contains also entries to allow delay information to be specified for each of the lines. These parameters are specific to every PCB design, the **PCB lengths** are contained on ZedBoard PCB trace length reports provided on the ZedBoard Hardware User Guide available in the zedboard.org webpage. Filling the lengths shown in the next figure will cause XPS to adjust delay parameters to achieve the best operation.



Pin Group	Length (mm)	Length (mils)	Package Length (mils)	Total Length (mils)	Propagation Delay (ps/inch)	Total Delay (ns)	DQS to CLK Delay (ns)	Board Delay (ns)
CLK0	55.77	2195.9	470	2665.9	160	0.427		
CLK1	55.77	2195.9	470	2665.9	160	0.427		
CLK2	41.43	1631.1	470	2101.1	160	0.336		
CLK3	41.43	1631.1	470	2101.1	160	0.336		
DQS0	51.00	2008.0	504	2512.0	160	0.402	0.025	
DQS1	50.77	1998.8	495	2493.8	160	0.399	0.028	
DQS2	41.59	1637.6	520	2157.6	160	0.345	-0.009	
DQS3	41.90	1649.4	835	2484.4	160	0.398	-0.061	
DQ[7:0]	50.63	1993.3	465	2458.3	160	0.393		0.410
DQ[15:8]	50.71	1996.4	480	2476.4	160	0.396		0.411
DQ[23:16]	40.89	1609.9	550	2159.9	160	0.346		0.341
DQ[31:24]	40.58	1597.8	780	2377.8	160	0.380		0.358

Figure 10: Delays in ZedBoard Hardware User Guide - 8 of Hardware User Guide

This information is used by Xilinx for **knowing** the **delay** of each component and be able to notify if there is an excessive delay which could cause latches or other errors. Once the above-mentioned configuration file of the peripherals is imported and the PCB lengths fill in, the result is the following figure.



☑ Enable DDR Controller

DDR Controller Configuration

Memory Type	DDR 3
Memory Part	MT41J128M16 HA-15E
Effective DRAM Bus Width	32 Bit
ECC	Disabled
Burst Length	8
Operating Freq (MHz)	533,333313
Internal Vref	☑
Operating Temperature(C)	Normal (0-85)

Training/Board Detail

DRAM Training: ☑ Write leveling ☑ Read gate ☑ Read data eye

	DQS3	DQS2	DQS1	DQS0
DQS to Clock Delay (ns)	-0,061	-0,009	0,027	0,025
	DQ[31:24]	DQ[23:16]	DQ[15:8]	DQ[7:0]
Board Delay (ns)	0,358	0,341	0,411	0,410

Memory Part Configuration

DRAM IC Bus Width	16 Bits
DRAM Device Capacity	2048 MBits
Speed Bin	DDR3_1066F
Bank Address Count (bits)	3
Row Address Count (bits)	14
Col Address Count (bits)	10
CAS Latency (cycles)	7
CAS Write Latency (cycles)	6
RAS to CAS Delay (cycles)	7
Precharge Time (cycles)	7
tRC (ns)	49,50
tRASmin (ns)	36,00
tFAW (ns)	45,00
Additive Latency (ns)	0

Pin Group	Length (mm)	Package Length (mils)	Propagation Delay (ps/inch)	DQS to CLK Delay (ns)	Board Delay (ns)
CLK0	55.77	470	160		
CLK1	55.77	470	160		
CLK2	41.43	470	160		
CLK3	41.43	470	160		
DQS0	51.00	504	160	0.025	
DQS1	50.77	495	160	0.027	
DQS2	41.59	520	160	-0.009	
DQS3	41.90	835	160	-0.061	
DQ[7:0]	50.63	465	160		0.410
DQ[15:8]	50.63	480	160		0.411
DQ[23:16]	40.89	550	160		0.341
DQ[31:24]	40.58	780	160		0.358

Figure 11: PS7 DDR Configuration after Importing the Configuration File

Finally XPS can be closed. After that, the top-level project design file “system.xmp” and the Microprocessors Hardware Specification file (*.mhs) are added to the PlanAhead project. This file is the hardware netlist of the processor subsystem which fully defines the embedded system hardware. Expanding the Embedded Design Sources in the Sources view displays the different target files.

Moreover, other **files** are **created**, for instance “ps7_init.c”, “ps7_init.h”, “ps7_init.tcl” and “ps7_init.html”, which contain:

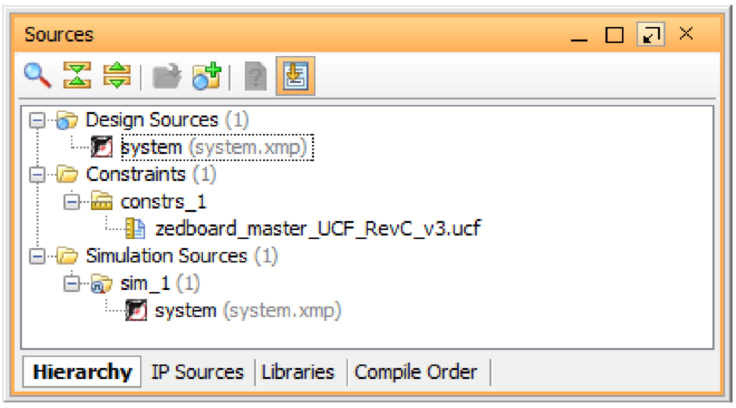
- On one hand, “**ps7_init.html**” has the documentation of register level details, use as a reference alternative to browsing through initialization source code. For example, the initialization data for:
  - Processing System.
  - PLLs.
  - Clocks.
  - DDR memory.
  - MIO.
- On the other hand, the **other files** contain the source files containing PS configuration setup. For instance:
  - List of the peripherals selected in the design.
  - List of IP blocks presented in the design.
  - The address map for processors Cortex-A9 0 and Cortex-A9 1.
  - The Multiplexed Input Output (MIO) configuration.
  - The Zynq-7000 Peripheral configuration.

It will be created also the **XML file**, which contains the processor and peripheral instantiation and addresses for FSBL and BSP generation.

3.3. Design Constraints

Xilinx knows the PS pin location mapping and the timing, based on the configuration file which has being imported in the XPS. Nevertheless, the PL needs a User Constraint File (UCF) to define the **pin locations and PS timing**, with the exception of circuitry driven from the PS fabric clock.

The ZedBoard ***.ucf file** can be downloaded on the ZedBoard official webpage and imported such as the above-mentioned PS7 configuration file.



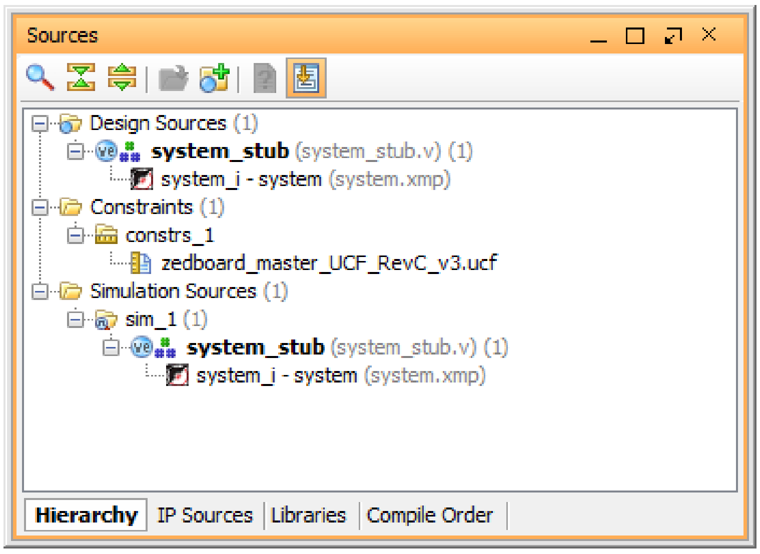
Design Sources (1)
system (system.xmp)
Constraints (1)
constrs_1
zedboard_master_UCF_RevC_v3.ucf
Simulation Sources (1)
sim_1 (1)
system (system.xmp)

Figure 12: Sources Window in PlanAhead after Importing the *.ucf File

3.4. Top HDL Module and Hardware Platform Building

Now, the hardware platform is completely configured. The configuration includes clock and DDR controller settings, it also enables and maps a UART peripheral. The hardware platform has to be built and exported to the Software Development Kit (SDK) for being able to develop any application. Building includes the top level wrapper generation, the synthesis, the implementation and the BitStream generation.

The **top level wrapper** (or Top HDL in PlanAhead) is a top level module for the design. PlanAhead generates a "system_stub.vhd" top-level module for the design where "system.xmp" (the embedded system) is now a sub-system of "system_stub".



Design Sources (1)
system_stub (system_stub.v) (1)
system_i - system (system.xmp)
Constraints (1)
constrs_1
zedboard_master_UCF_RevC_v3.ucf
Simulation Sources (1)
sim_1 (1)
system_stub (system_stub.v) (1)
system_i - system (system.xmp)

Figure 13: Sources Window with the Top HDL Module Generated

The next processes which are needed are the synthesis, implementation, verification, and BitStream generation. An elaborated, detailed **schema** of this whole process is showed in the next figure.

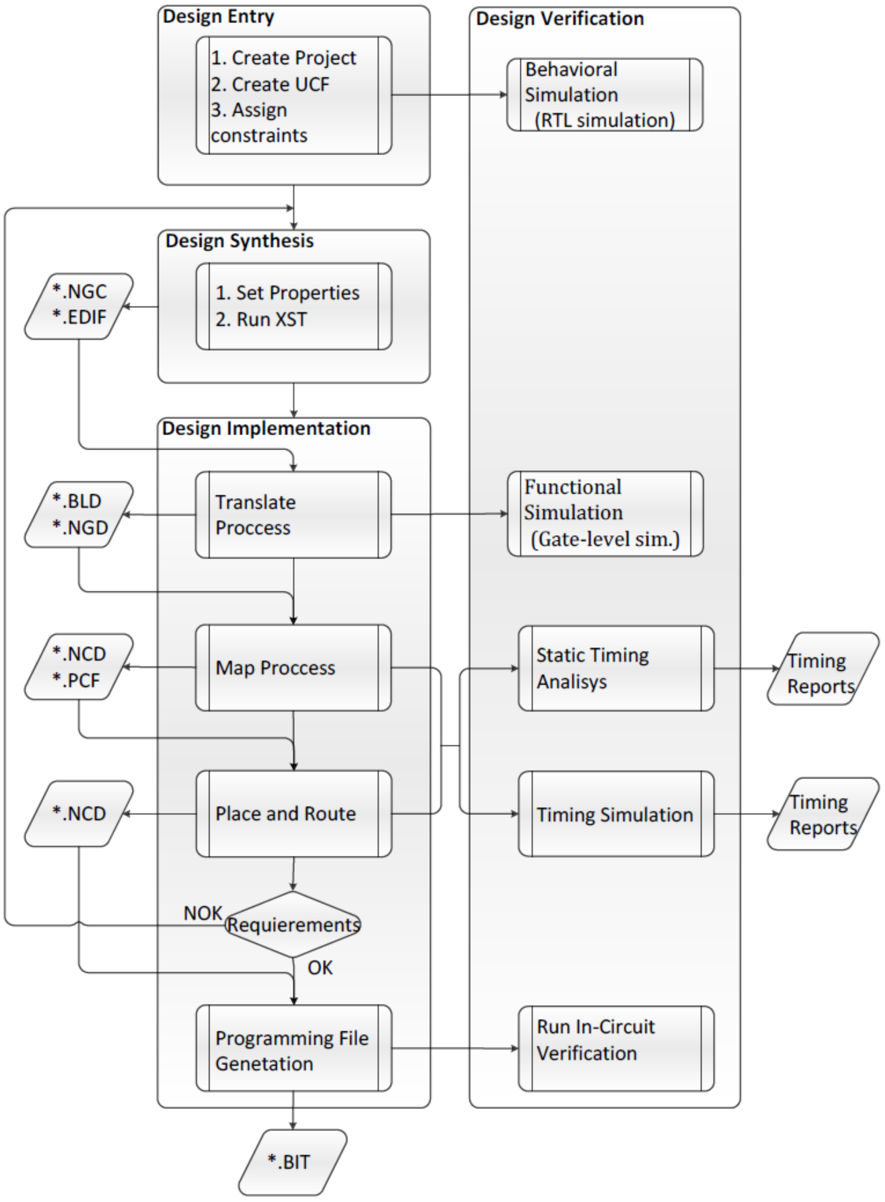


Figure 14: Design Flow Schema

The BitStream generation finishes the hardware design. Any **software** project associated with the hardware design has to be created within SDK (e.g. the First Stage Boot Loader or the Boot file which will be copy into the SD Card). Therefore, the project must be exported to SDK. After launching SDK, PlanAhead can be closed, it will not further required.

In addition to the BitStream file, PlanAhead also **exports** the Hardware Platform Specification for the design, “system.xml” **to SDK**, which is opened by default when SDK is launched. This file contains the hardware platform description for FSBL and BSP generation. Four more files are exported to SDK, the already mentioned files “ps7_init.c”, “ps7_init.h”, “ps7_init.tcl” and “ps7_init.html”. These settings are used by SDK when adding and mapping the low level drivers for the peripherals selected and for initializing the processing system so that the applications can be run on top of the processing system.

Once the basic ZedBoard project is built, any program can be developed and implemented on ZedBoard (using an Operating System or with a Standalone configuration). Therefore, any version of Linux with this hardware configuration can be developed and implemented continuing from this point.

Finally, note that it is strongly recommended to perform the “Appendix 1” up to this point before continuing reading this chapter.

3.5. Stage 0 or BootROM

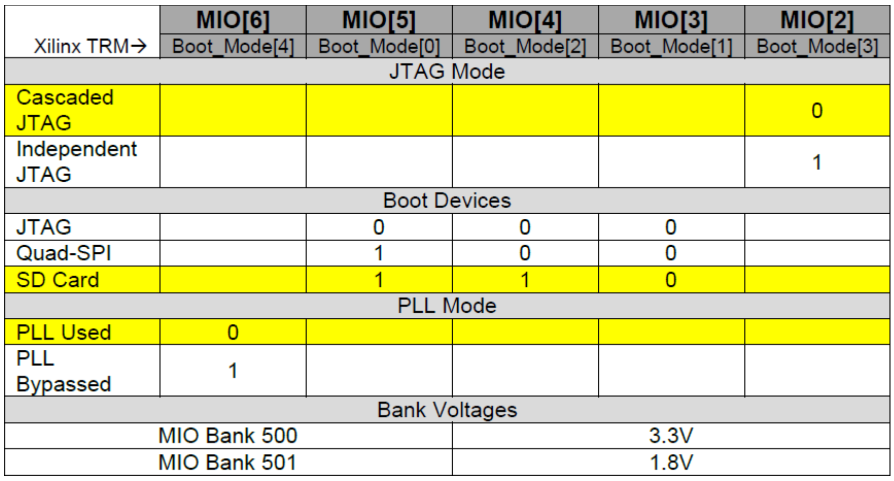
Zynq-7000 AP SoC devices use a **multi-stage boot process** that supports both non-secure and secure boot. The PS is the master of the boot and configuration process. This is the only no-user-configurable stage.

Upon reset, the device mode pins are read to **determine** the **primary boot device** to be used: NOR, NAND, Quad-SPI, SD Card or JTAG; depending on the chosen configuration. The boot mode pins are the MIO pins from 2 to 8. Therefore, these **MIO pins** are used as follows:

- Pin 2 of MIO or pin 3 of Boot_Mode: sets the JTAG mode.
- Pins from 3 to 5 of MIO or pins from 0 to 2 of Boot_Mode: select the boot mode.
- Pin 6 of MIO or pin 4 of Boot_Mode: enables the internal PLL.
- Pins from 7 to 8 of MIO or Pins from 0 to 1 of Vmode: are used to configure the I/O bank voltages, however these are fixed on ZedBoard and not configurable.

The ZedBoard provides three **jumpers** for the MIO configuration (from pin 2 to 6). These jumpers allow users to change the mode options, including using cascaded JTAG configuration as well as using the internal PLL.

The Linux OS will be introduced on ZedBoard using the SD Card; hence the MIO **Configuration** Modes are the list below, with the required setting highlighted in yellow:



	MIO[6]	MIO[5]	MIO[4]	MIO[3]	MIO[2]
Xilinx TRM→	Boot_Mode[4]	Boot_Mode[0]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

Figure 15: ZedBoard Configuration Modes to Run Linux

Looked at a different way, the jumpers must be fixed as the following image:

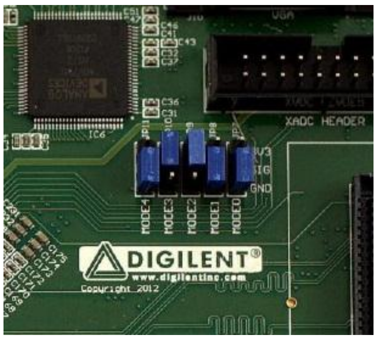


Figure 16: ZedBoard SD Card Boot Mode Jumper Setting

Therefore, when the board is switched on, it automatically detects the configured boot mode and it will load the First Stage Boot Loader (FSBL) which will be available in the SD Card. Then the FSBL will take control and will prepare the system so a larger boot loader will be able to be loaded (U-Boot).

3.6. Boot Image, "boot.bin"

Once the BootROM is configured, the next step is creating the **boot file**, which is a binary composite image, **responsible for**:

- Board initialization using the PS configuration data provided by XPS.
- Programming the PL of the FPGA with the Hardware BitStream (if it exists, this step is optional, and it can be performed later).
- Loading the Operating System (OS) Image or a Standalone (SA) Image or the Second Stage Boot Loader image and starting executing it. In this project the Second Stage Boot Loader (U-Boot) will be loaded.

It supports **multiple partitions**, and each partition can be a code image or a BitStream. In this case, it will be only consisted on 3 files:

- The specific **First Stage Boot Loader** (FSBL) for ZedBoard.
- The specific **Programmable Logic Hardware BitStream** file for ZedBoard (optional).
- The U-Boot file, as **Second Stage Boot Loader**.

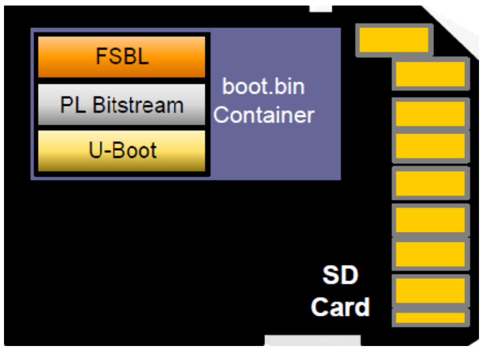


Figure 17: Boot Image Container

First, the **FSBL**, together with the **BitStream** file, is responsible for initializing the processor resources so a larger boot loader can be loaded (in this case U-Boot). It can be generated directly by the SDK project template. Among its **features** highlight the following:

- Initializing the Zynq PS (PLLs, DDR memory controller, MIO, and UART).
- Configuring the PL with the Programmable Logic Hardware BitStream (optional).
- Loading the application code from the boot medium to memory.
- Transferring the execution to the application code (U-Boot).

Second, the **Second Stage Boot Loader** file loads the kernel and passes the device tree to Linux. U-Boot will be used as Second Stage Boot Loader. It is an open source software project U- which performs important boot functions:

- Initializes the platform hardware needed to load kernel.
- Loads Linux kernel from boot medium to main memory (DDR3).
- Starts the Linux kernel with specified boot parameters.

It is mostly used to load and boot a kernel image, but it also allows developers to change the kernel image and the root file system stored in flash. Files can be exchanged between the target and the development workstation.

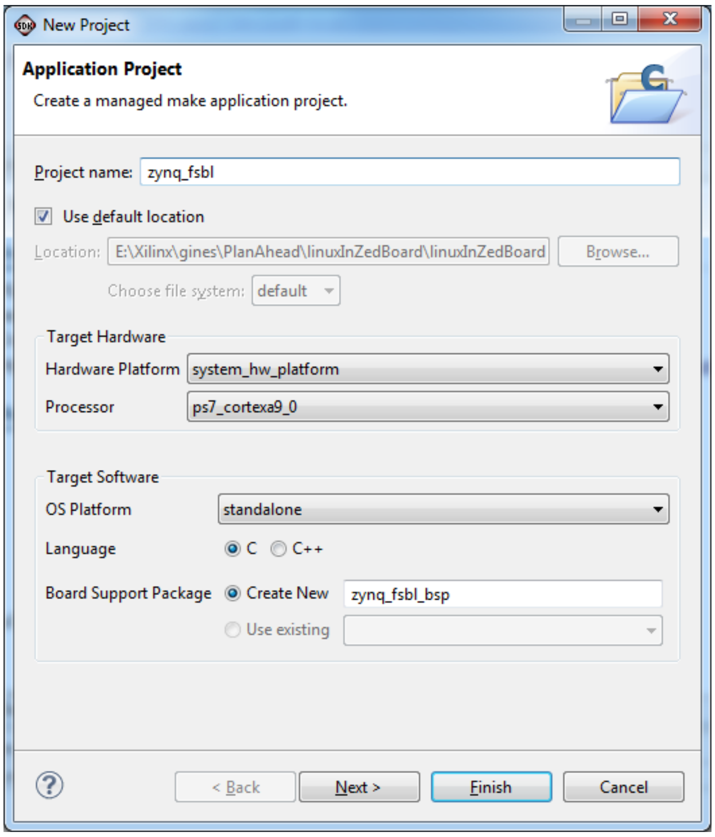
3.6.1. First Stage Boot Loader (FSBL)

Once the BootROM is configured, the next step is creating the FSBL. Remember that the FSBL, together with the BitStream file, is responsible to initialize the processor resources so a larger boot loader can be loaded (in this case U-Boot). It can be generated directly by the SDK project template. Among its **features** highlight the following:

- Initializing the Zynq PS (PLLs, DDR memory controller, MIO, and UART).
- Loading the application code from the boot medium to memory.
- Transferring the execution to the application code (U-Boot).
- Configuring the PL with the Programmable Logic Hardware BitStream together with the BitStream file (optional).

Finally, it is important to know that the First Stage Boot Loader file, as the BitStream file, is always **related to** the **hardware** architecture and configuration, regardless of whether an OS is used. Therefore, they will not depend on the Linux OS version to implement on ZedBoard. Also for this reason, the OS Platform is selected as "standalone". Nevertheless, this file will not be the same file for the Ubuntu Linux OS version which will be developed in "Chapter 4: Ubuntu Linux OS on ZedBoard", because this file also depends on the hardware configuration, which will be different in the next chapter (e.g. it will be included a HDMI monitor, a USB mouse, and a USB keyboard, among others).

The Xilinx tool Software Development Kit (SDK) is used to create the FSBL. After open SDK in the same path directory of the created project of the last subchapter, a new Application Project must be selected to build the FSBL and choose the "Zynq FSBL" mode.



New Project

Application Project

Create a managed make application project.

Project name: zynq_fsbl

[✓] Use default location

Location: E:\Xilinx\gines\PlanAhead\linuxInZedBoard\linuxInZedBoard

Choose file system: default

Target Hardware

Hardware Platform: system_hw_platform

Processor: ps7_cortexa9_0

Target Software

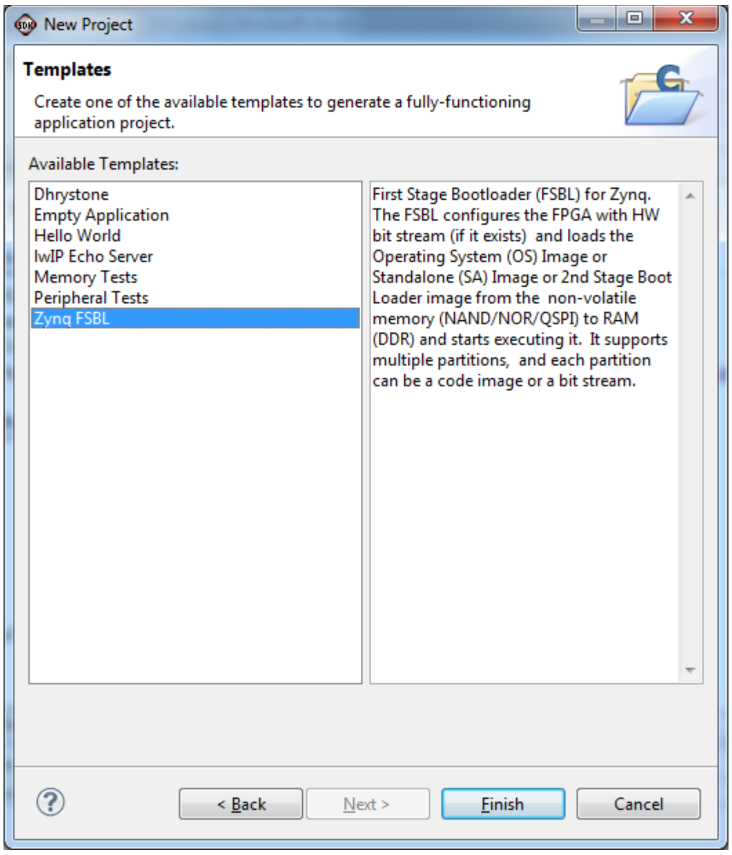
OS Platform: standalone

Language: [✓ C] C++

Board Support Package: [✓ Create New] zynq_fsbl_bsp

Use existing

Figure 18: New Project Window to Create the FSBL in SDK 1/2



New Project

Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

- Dhrystone
- Empty Application
- Hello World
- lwIP Echo Server
- Memory Tests
- Peripheral Tests
- [✓ Zynq FSBL]

First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Figure 19: New Project Window to Create the FSBL in SDK 2/2

It is not necessary a better **explanation** for the **FSBL** that the provided by Xilinx in the latest figure.

"First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream."

In addition, a few comments about these last 2 figures are required in order to achieve a better understanding of the FSBL.

First, the **"standalone" mode** has been chosen for the "OS Platform" instead of the other available option, "linux". It may seem contradictory, given that it is sought to build a Linux OS. Nevertheless, as already remarked, the FSBL don"t depend on the Operating System which will be used, but only of the hardware target where it is implemented on.

For this purpose, it is required to configure the "**Target Hardware**", where will be selected the hardware platform configured in the previous subchapter, together with the first processor of the ZedBoard, because of its greater speed over the second processor.

Once the file is successfully built, it can be directly copied of its path directory which is showed in the image below. Note that this file can be directly opened with SDK, allowing a better understanding of it.

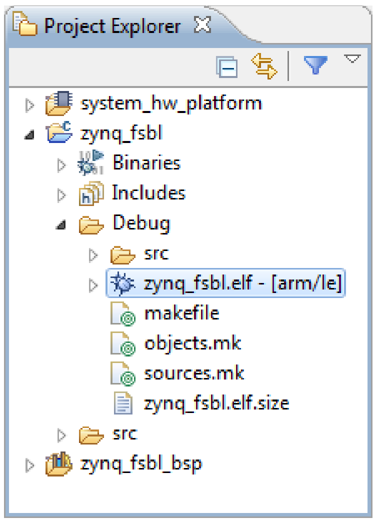

Project Explorer
- system_hw_platform
- zynq_fsbl
  - Binaries
  - Includes
  - Debug
    - src
    - zynq_fsbl.elf - [arm/le]
    - makefile
    - objects.mk
    - sources.mk
    - zynq_fsbl.elf.size
  - src
- zynq_fsbl_bsp

Figure 20: SDK Project Explorer Window after Building the FSBL File

3.6.2. Programmable Logic Hardware BitStream file

The Programmable Logic Hardware BitStream file has been mentioned and created during this chapter, but it has not been necessary up to now. It is therefore the ideal moment to talk about it and about its function.

If its **definition** of "BitStream" is looked for, the result is the following.

"A BitStream or bit stream is a time series or sequence of bit".

Therefore, the PL Hardware BitStream file is a sequence of bits which contains the Hardware and PL configuration for a specific board, such as the ZedBoard, in a binary language that the specific board can understand and implement.

As above mentioned, the BitStream file, together with the FSBL file, is responsible for **initializing** the processor resources, but the PL Hardware BitStream file is focus in the **Programmable Logic Hardware**, while the FSBL file also initializes the PS (PLLs, DDR memory controller, MIO, and UART).

Remember that this file **is optional**. If the PL is not required, this file is not necessary at all. Moreover, the PL Hardware BitStream can also be added manually later.

Finally, note that it can be found in the SDK project or in the PlanAhead project.

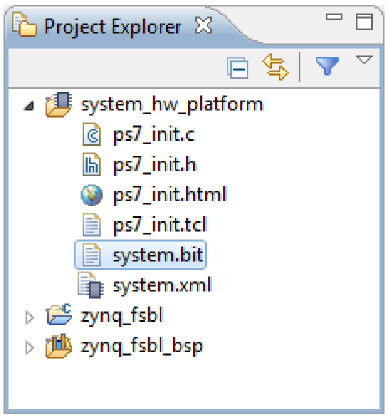


Figure 21: SDK Project Explorer Window with the PL Hardware BitStream

3.6.3. Second Stage Boot Loader, U-Boot

Most of the required files to build the boot file have been obtained. However, one more file is necessary, the Second Stage Boot Loader. This stage loads the Linux Kernel and passes the device tree to Linux. U-Boot will be used as Second Stage Boot Loader.

The **U-Boot Universal Boot-Loader** is a GPL cross-platform boot loader pioneered by project leader Wolfgang Denk but forged by developers and user community. U-Boot provides out-of-the-box support for hundreds of embedded boards and a wide variety of CPU architectures including ARM. The development of U-Boot is closely related to Linux, some parts of the source code originate in the Linux source tree, and they have some header files in common. In addition, special provision has been made to support the Linux image booting.

U-Boot carries out different kind of functions. On one hand, it performs important **boot functions**, such as…

- Initialize the platform hardware needed to load the Linux Kernel.
- Load the Linux Kernel from boot medium to the main memory (DDR3).
- Start the Linux Kernel with the specified boot parameters.

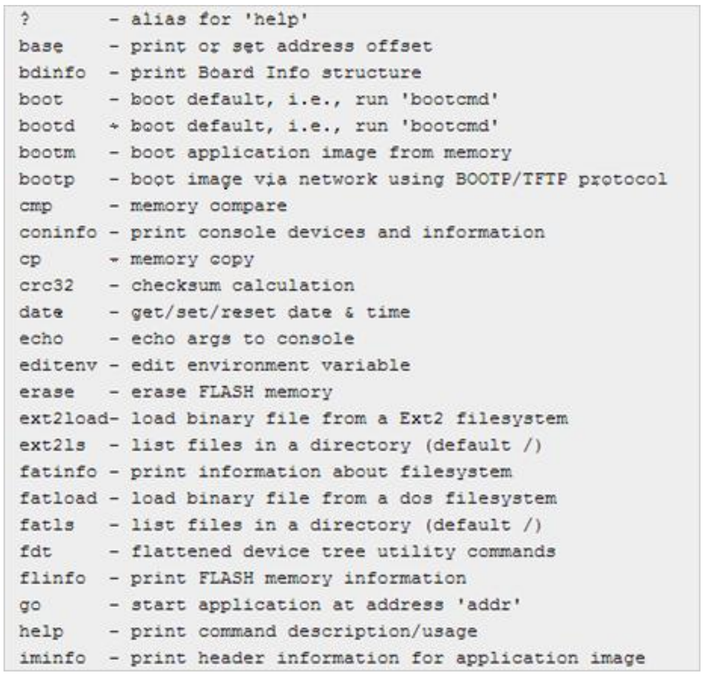
On the other hand, it also provides some convenient **features** for the development environment, like…

- Provide network access.
  - Ping IP addresses.
  - Download binary images via TFTP.
- Allow memory test, copy, and comparison.
  - Reads and writes arbitrary memory locations.
  - Copies binary images from one location in memory to another.
- Configure and access to the hardware peripheral devices directly (e.g. Quad-SPI Flash, I2C, and Ethernet).
- Detect the boot mode and run related boot macros.

U-Boot can **obtain** the **Kernel Image** from a SD Card, partitioned QSPI Flash, and even through Ethernet using TFTP (having a functional TFTP server).

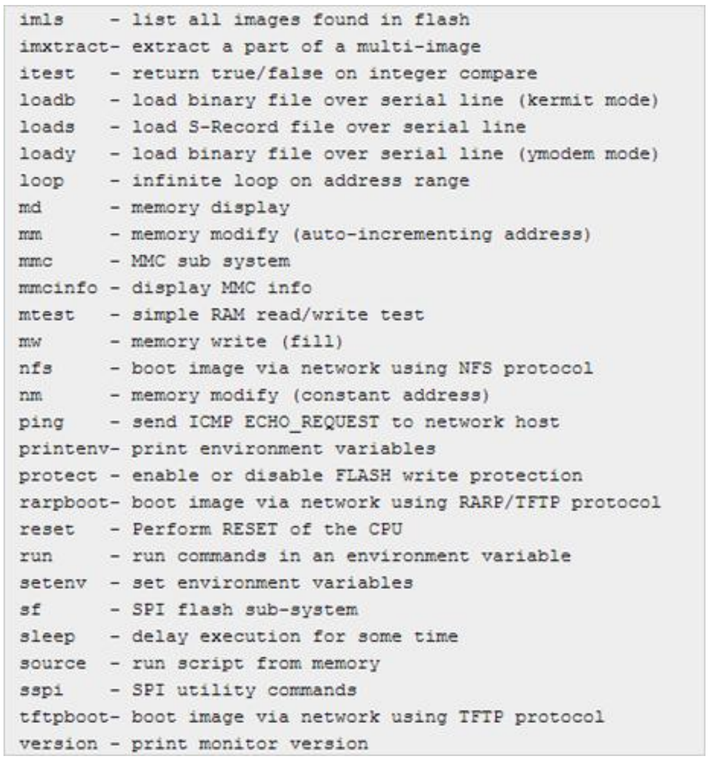
By default, if no key is pushed after switch on ZedBoard, U-Boot starts the “autoboot” procedure, which looks for BootMode pins settings again for the source of the Kernel Image, the Device Tree Blob and the Root File System Image.

Nevertheless, if the auto-boot is stopped by pushing a key, U-boot provides a **console interface** to execute commands, which can be used for the above mentioned purposes.



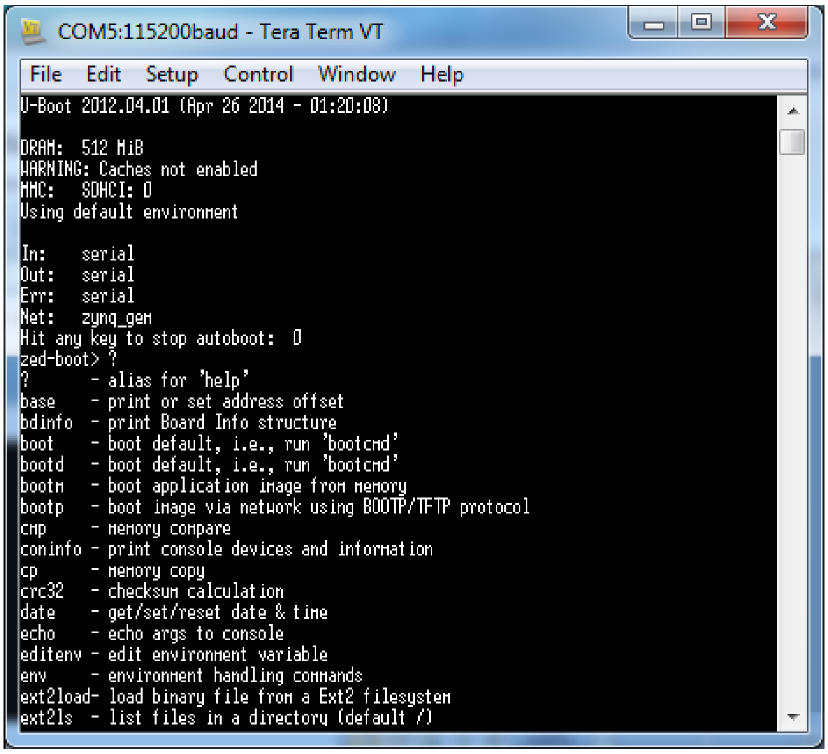
```
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
date    - get/set/reset date & time
echo    - echo args to console
editenv - edit environment variable
erase   - erase FLASH memory
ext2load- load binary file from a Ext2 filesystem
ext2ls  - list files in a directory (default /)
fatinfo - print information about filesystem
fatload - load binary file from a dos filesystem
fatls   - list files in a directory (default /)
fdt     - flattened device tree utility commands
flinfo  - print FLASH memory information
go      - start application at address 'addr'
help    - print command description/usage
iminfo  - print header information for application image
```

Figure 22: U-Boot Commands 1/2



```
imls    - list all images found in flash
imxtract- extract a part of a multi-image
itest   - return true/false on integer compare
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loady   - load binary file over serial line (ymodem mode)
loop    - infinite loop on address range
md      - memory display
mm      - memory modify (auto-incrementing address)
mmc     - MMC sub system
mmcinfo - display MMC info
mtest   - simple RAM read/write test
mw      - memory write (fill)
nfs     - boot image via network using NFS protocol
nm      - memory modify (constant address)
ping    - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
setenv  - set environment variables
sf      - SPI flash sub-system
sleep   - delay execution for some time
source  - run script from memory
sspi    - SPI utility commands
tftpboot- boot image via network using TFTP protocol
version - print monitor version
```

Figure 23: U-Boot Commands 2/2


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
U-Boot 2012.04.01 (Apr 26 2014 - 01:20:08)

DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   SDHCI: 0
Using default environment

In:    serial
Out:   serial
Err:   serial
Net:   zynq_gem
Hit any key to stop autoboot:  0
zed-boot> ?
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
date    - get/set/reset date & time
echo    - echo args to console
editenv - edit environment variable
env     - environment handling commands
ext2load- load binary file from a Ext2 filesystem
ext2ls  - list files in a directory (default /)
```

Figure 24: U-Boot Command Console Running on ZedBoard

For all these reasons, U-Boot is the ideal Second Stage Boot-Loader for the ZedBoard. There are **three different methods** to get the desired file "u-boot.elf" and the three provide exactly the same result and create the same "u-boot.elf" file.

- Download the official **Xilinx U-Boot repository**, configure it for the ZedBoard and build the required file.
- Download the **U-Boot BSP generator** for the Xilinx git repository, add it to Xilinx SDK and built the required file.
- Configure **BuildRoot** to automatically download and configure a specific U-Boot repository and to build the file (recommended).

The three methods will be explained in this chapter in order to better understanding about U-Boot. Nevertheless, only one of these methods is necessary to develop. Due to its greater simplicity and convenience, the third method is the recommended one.

From now, a **Linux OS** will be **necessary** to continue. Regardless of whether a Linux OS is available in the computer, the recommended solution is installing the CentOS OS in the Virtual Machine VMware Player. Anyway, any other Virtual Machine is also suitable.

CentOS (abbreviated from Community Enterprise Operating System) is a Linux distribution that attempts to provide a free, enterprise class, community-supported computing platform which aims to be 100% binary compatible with its upstream source, Red Hat Enterprise Linux (RHEL). This OS is recommended because it supports the AMD architecture.

How must be installed and configured the program is showed in "Appendix 2: Prerequisites".

The **first method**, download the official Xilinx U-Boot repository, is the traditional way to create the "u-boot.elf" file. After downloading the source code for U-Boot of the official Xilinx U-Boot repository, it can be opened and configured in the Linux "Terminal" program.

However, before building the U-Boot file, a **cross compiler** is required, which is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. In this case, the **Sourcery Codebench** will be used. It is a complete development environment for embedded C/C++ development on ARM, among others. Sourcery CodeBench includes:

- GNU C and C++ compilers.
- GNU assembler and linker.
- C and C++ runtime libraries.
- GNU debugger.

Once the Cross Compiler is downloaded and configured (see "Appendix 2: Prerequisites"), U-Boot can be built for one specific platforms using the "Terminal" command windows of a Linux OS.

In order to **configure U-Boot** for a specific hardware platform, the command "make <u-boot target>_config" is required. In this case, to configure U-Boot for the ZedBoard, the exact command is "make zynq_zed_config". It configures the U-Boot source tree with the appropriate soft links to select ARM as the target architecture, the ARM v7, the Zynq SoC and the ZedBoard as the target platform. With the environment variables properly set for the cross-compiling toolchain, U-Boot will be built for the Zynq ARM architecture after executing the “make” command which will generate the “u-boot” file. Finally, it should be looked in the “u-boot-xlnx” folder and renamed as “u-boot.elf”.

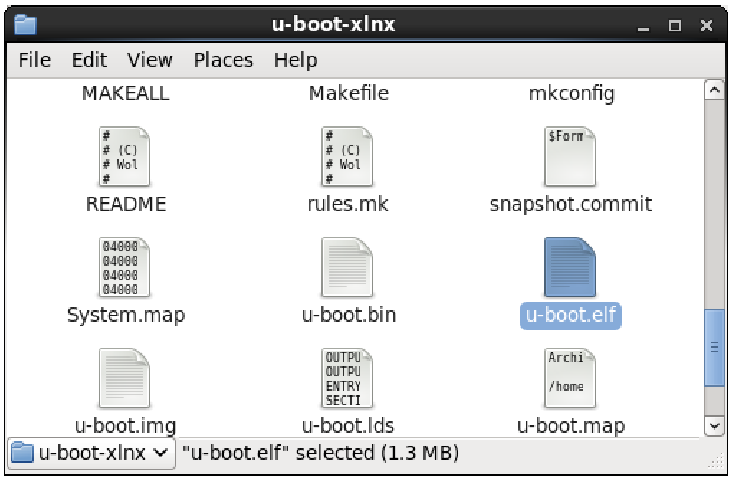
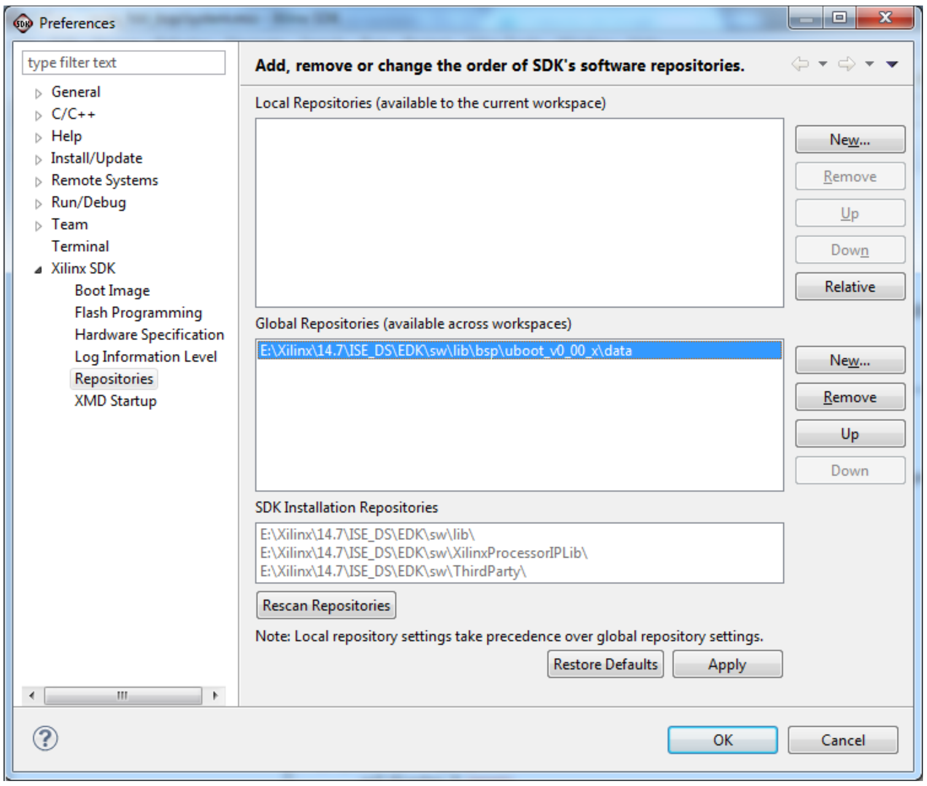


Figure 25: U-Boot Directory in Linux with the “u-boot.elf” File

Otherwise, if the **second method** is chosen, the U-boot BSP generator for Xilinx SDK must be downloaded and added to Xilinx SDK as a new repository.



Add, remove or change the order of SDK's software repositories.

Local Repositories (available to the current workspace)

Global Repositories (available across workspaces)

E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\bsp\uboot_v0_00_x\data

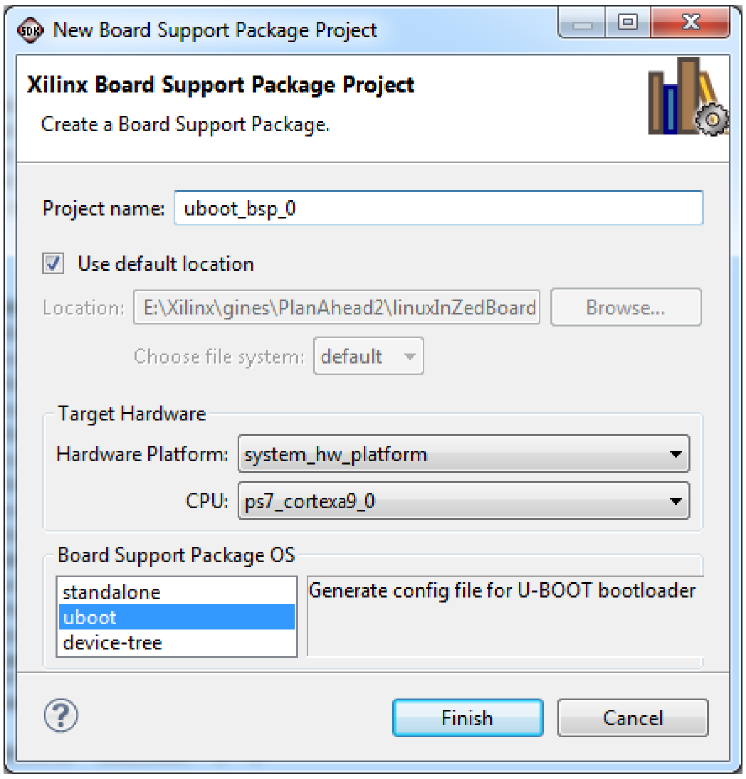
SDK Installation Repositories

E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\
E:\Xilinx\14.7\ISE_DS\EDK\sw\XilinxProcessorIPLib\
E:\Xilinx\14.7\ISE_DS\EDK\sw\ThirdParty\

Note: Local repository settings take precedence over global repository settings.

Figure 26: Adding a Board Support Package in Xilinx SDK 1/2

Note that also the Device Tree Generator can be added, it will be discussed in the following section. After adding the U-Boot repository, the required .elf file can be created after creating a new Board Support Package choosing the “uboot” option.


New Board Support Package Project

Xilinx Board Support Package Project

Create a Board Support Package.

Project name: uboot_bsp_0

[✓] Use default location

Location: E:\Xilinx\gines\PlanAhead2\linuxInZedBoard

Choose file system: default

Target Hardware

Hardware Platform: system_hw_platform

CPU: ps7_cortexa9_0

Board Support Package OS

standalone

[✓ uboot]

device-tree

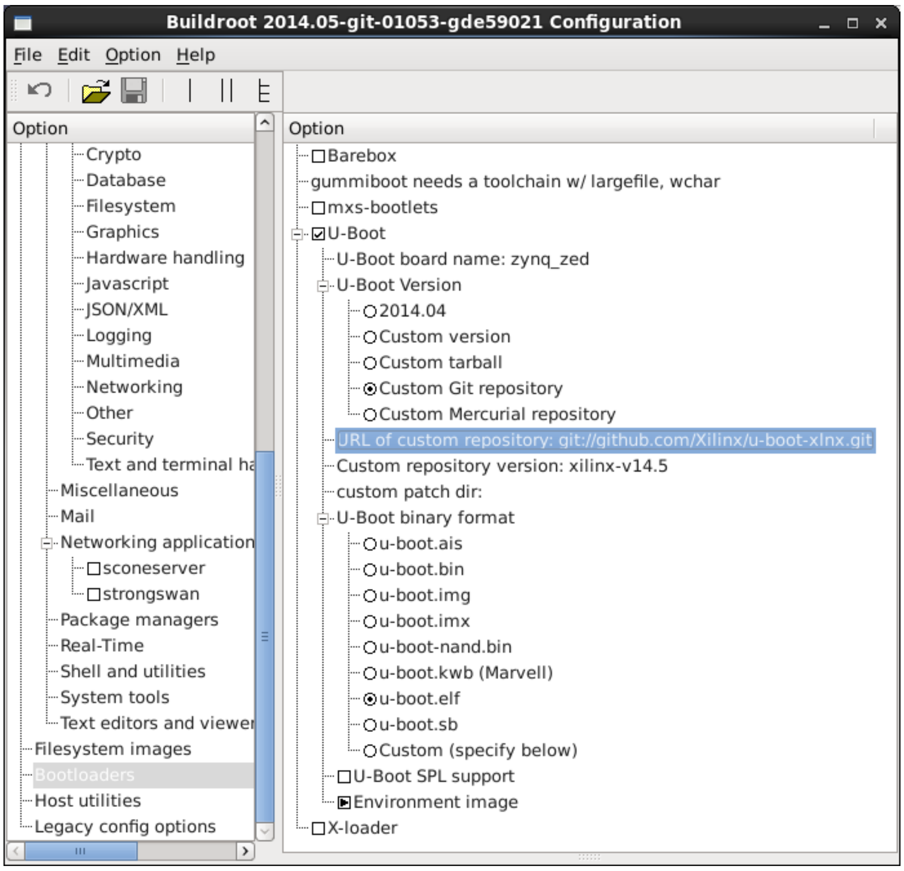
Generate config file for U-BOOT bootloader

Figure 27: Creating the U-Boot Board Support Package

Finally, the **last method**, building U-Boot using the BuildRoot tool, is the recommended method. The reasons to recommend it are already mentioned, **BuildRoot** allows not only configure the U-Boot file, but also the Device Tree Binary, the Linux Kernel and the Root File System Image. Thus, it avoid wasting time in downloading each required git directory for each one and setting them separately, when all of them can be configured by a single tool.

In terms of U-Boot, it has a whole **setting window** in BuildRoot, where the options which were explained in the first method are ready to be configured. It is required…

- The Board’s name.
- The U-Boot version.
- The URL of the custom repository and its version, after selecting the option “Custom Git repository”.
- The U-Boot binary format (.elf in this case).



Buildroot 2014.05-git-01053-gde59021 Configuration

File Edit Option Help

```
Option
        Crypto
        Database
        Filesystem
        Graphics
        Hardware handling
        Javascript
        JSON/XML
        Logging
        Multimedia
        Networking
        Other
        Security
        Text and terminal ha
      Miscellaneous
      Mail
      Networking application
        ☐sconeserver
        ☐strongswan
      Package managers
      Real-Time
      Shell and utilities
      System tools
      Text editors and viewer
    Filesystem images
    Bootloaders
    Host utilities
    Legacy config options
```

```
Option
☐Barebox
gummiboot needs a toolchain w/ largefile, wchar
☐mxs-bootlets
☑U-Boot
  U-Boot board name: zynq_zed
  U-Boot Version
    ○2014.04
    ○Custom version
    ○Custom tarball
    ◉Custom Git repository
    ○Custom Mercurial repository
  URL of custom repository: git://github.com/Xilinx/u-boot-xlnx.git
  Custom repository version: xilinx-v14.5
  custom patch dir:
  U-Boot binary format
    ○u-boot.ais
    ○u-boot.bin
    ○u-boot.img
    ○u-boot.imx
    ○u-boot-nand.bin
    ○u-boot.kwb (Marvell)
    ◉u-boot.elf
    ○u-boot.sb
    ○Custom (specify below)
  ☐U-Boot SPL support
  ▣Environment image
☐X-loader
```

Figure 28: U-Boot Configuration Window in BuildRoot

Note that U-Boot is not the only **possible Second Stage Boot Loader**; there are also others such as Barebox, mxs-bootlets or X-loader. If one of them is selected, its respective window is displayed. In addition, there is a configuration file for BuildRoot which loads the default configuration automatically for ZedBoard, and includes this U-Boot configuration. It can be loaded by simply typing the command “make zedboard_defconfig”.

As already mentioned, the **three methods** provide the same result, then BuildRoot will be the best option to build the desired u-boot.

Occasionally the **configuration file** has to be **edited**. In this case no edition will be made because the default configuration is sufficient. Nevertheless, for further use, the file is going to be briefly explained. U-Boot is configured using configuration variables defined in a board-specific header file. They have two forms, configuration options and configuration settings. On one hand, the first ones are selected using macros in the form of CONFIG_XXX. On the other hand, the second ones are selected using macros in the form of CONFIG_SYS_XXXX.

U-Boot configuration is driven by a **header file** dedicated for a specific platform which contains the appropriate configuration and settings for this platform. The source tree includes a directory where these board-specific configuration header flies reside: “…/u-boot-xlnx/include/configs” or “…/buildroot/output/build/uboot-xilinx-v14.5/include/configs”, depending of the method used.

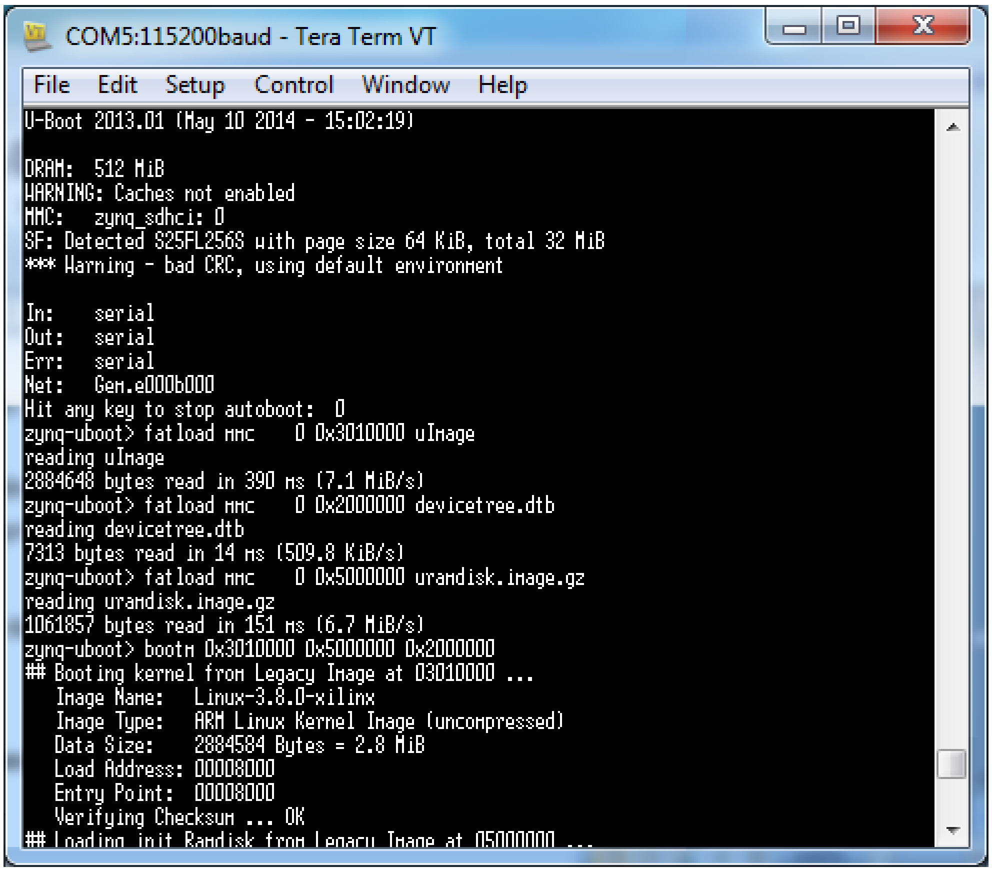
Numerous of features and modes of operation can be selected by adding definitions to the board-configuration header file called “zynq_zed.h” available in this folder. On one hand, “sdboot” is used to deal this ZedBoard system. On the other hand, “mmcinfo” is used to initialize the SD Card. Therefore, fetches inside the memory can be done. In this case, the **procedure** is as follows:

1. “sdboot” reads the Kernel Image uImage from the FAT partition and copies it into the DDR memory.
2. It reads the Device Tree Binary (.dtb) file and also loads it into memory.
3. Finally, it reads the compressed Root File System Image “uramdisk.image.gz” and also loads it.

After loading these files, U-Boot begins the execution at the RAM address where “uImage” is located.

Note that the **memory addresses** where these files are loaded can be changed in 2 ways. On one hand, it can be changed by modifying this file. On the other hand, by the U-Boot command window after switching on the board.

For instance, after loading the “uImage” and the Device Tree Binary, if the “uramdisk.image.gz” is too big, maybe it can be loaded overlapping and overwriting them. One possible solution can be to assign more memory between this file and the others.

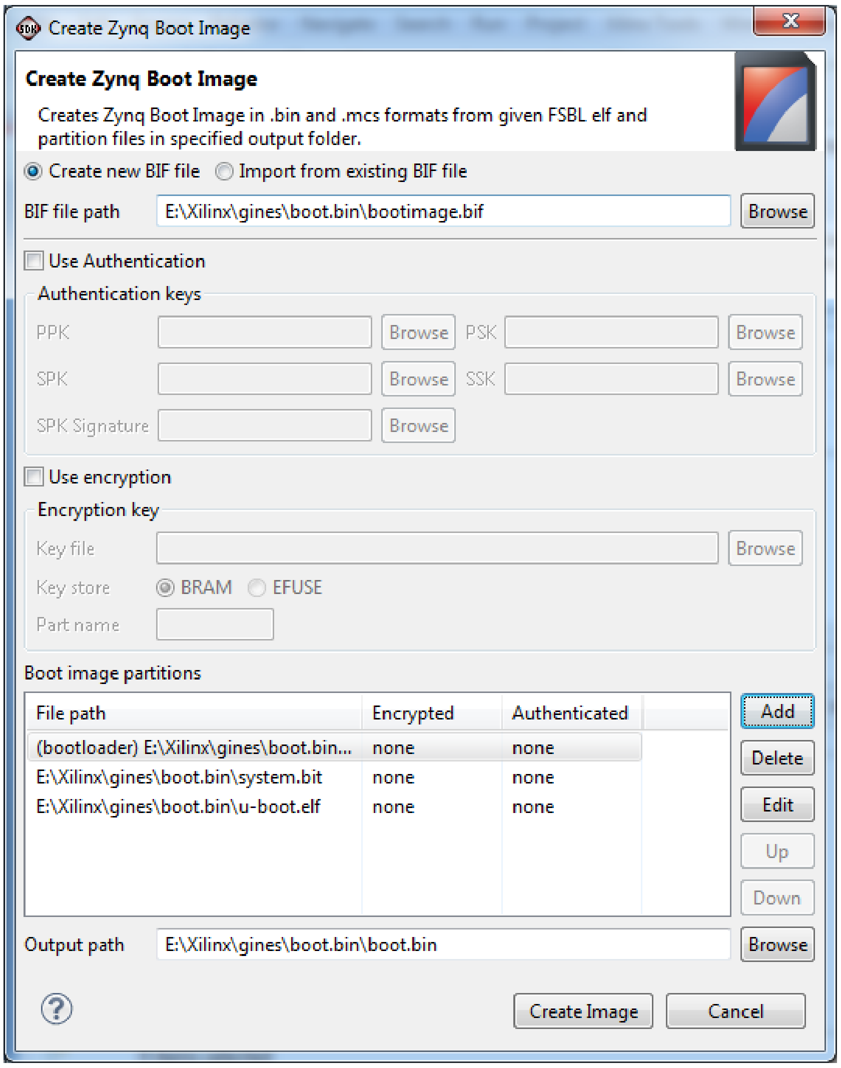


```
U-Boot 2013.01 (May 10 2014 - 15:02:19)

DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   zynq_sdhci: 0
SF: Detected S25FL256S with page size 64 KiB, total 32 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   Gem.e000b000
Hit any key to stop autoboot:  0
zynq-uboot> fatload mmc     0 0x3010000 uImage
reading uImage
2884648 bytes read in 390 ms (7.1 MiB/s)
zynq-uboot> fatload mmc     0 0x2000000 devicetree.dtb
reading devicetree.dtb
7313 bytes read in 14 ms (509.8 KiB/s)
zynq-uboot> fatload mmc     0 0x5000000 uramdisk.image.gz
reading uramdisk.image.gz
1061857 bytes read in 151 ms (6.7 MiB/s)
zynq-uboot> bootm 0x3010000 0x5000000 0x2000000
## Booting kernel from Legacy Image at 03010000 ...
   Image Name:   Linux-3.8.0-xilinx
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2884584 Bytes = 2.8 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 05000000 ...
```

Figure 29: Changing the Memory Address Load Location on ZedBoard

3.6.4. Generating "boot.bin" file

All the required files to build the boot file have been obtained. Therefore, the boot image can be created using the "**Bootgen**" tool which is integrated into SDK.


Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

[✓ Create new BIF file] Import from existing BIF file

BIF file path: E:\Xilinx\gines\boot.bin\bootimage.bif

Use Authentication (unchecked)

Use encryption (unchecked)

Key store: [✓ BRAM] EFUSE

Boot image partitions

File path	Encrypted	Authenticated
(bootloader) E:\Xilinx\gines\boot.bin...	none	none
E:\Xilinx\gines\boot.bin\system.bit	none	none
E:\Xilinx\gines\boot.bin\u-boot.elf	none	none

Output path: E:\Xilinx\gines\boot.bin\boot.bin

Figure 30: Zynq Boot Image Generation Window in SDK

For adding the Boot image partitions, it is **important** to know that the **order** of the images should always be the same: first, the First Stage Boot Loader ("zynq_fsbl.elf"); second, the Programmable Logic BitStream ("system.bit"); and finally, the software application file, in this case, the Second Stage Boot Loader U-Boot ("u-boot.elf"). The reason is that the ZedBoard need to be booted in this order.

The Stage 0 or BootROM is able to boot the processor from several different non-volatile memory types, but requires a data structure referred to as the Boot Image Format File (BIF) to obtain instructions about how analyze the different boot components. The BIF file specifies each component of the boot image, ordered by boot-sequence, and allows optional attributes to be applied to each image component.

"**Bootgen**" is a standalone tool to create a bootable image appropriate for ZedBoard. The program assembles the boot image by merging the BIT and ELF files into a single boot image with the binary output file "boot.bin" format to be loaded into Zynq devices at boot time. For its part, it is also generated a bootimage.bif file with the format to define which files are integrated and what order they are added to the binary output file (only used by SDK). Optionally, it is possible to encrypt and authenticate each partition.

The generated file "boot.bin" will be the first file to introduce on the SD Card which will be introduced in the SD Slot of the ZedBoard.

3.7. Device Tree Binary, "devicetree.dtb"

The **Linux Kernel** is a piece of embedded standalone software running on hardware. It provides a standardized interface for programmers to utilize all hardware resources without knowing the details. Thus, it has to **know** every **detail** about the hardware where it is running on. The Linux Kernel uses the data structure called "**Device Tree Blob**" or "Device Tree Binary (DTB)" to describe the hardware.

The **Device Tree Binary** is a database which represents the hardware components on a given board and has been chosen as the default mechanism to pass low-level hardware information from the Boot Loader to the Kernel. In the same way that U-Boot or other low-level firmware, being able to master the DTB requires complete knowledge of the underlying hardware.

This file is build from the **Device Tree Source** (DTS) file. The DTS file is the DTB file write in a human-editable format. This file is usually provided as part of the Linux Kernel source tree, but if some custom hardware is done, it has to be also customized; hence, the DTB file has to be edited too.

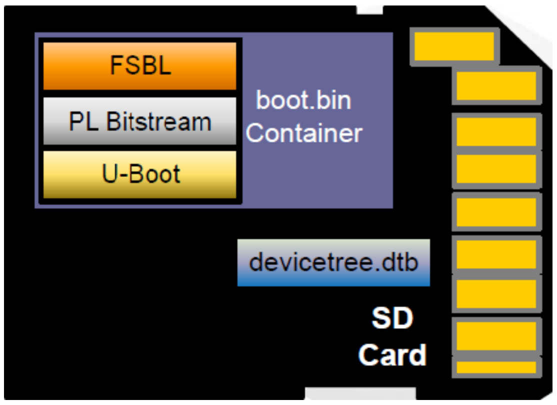


Figure 31: The "devicetree.dtb" file

3.7.1. Device Tree Source (DTS File)

Similar to U-Boot, there are the same **three methods** to get the DTS file.

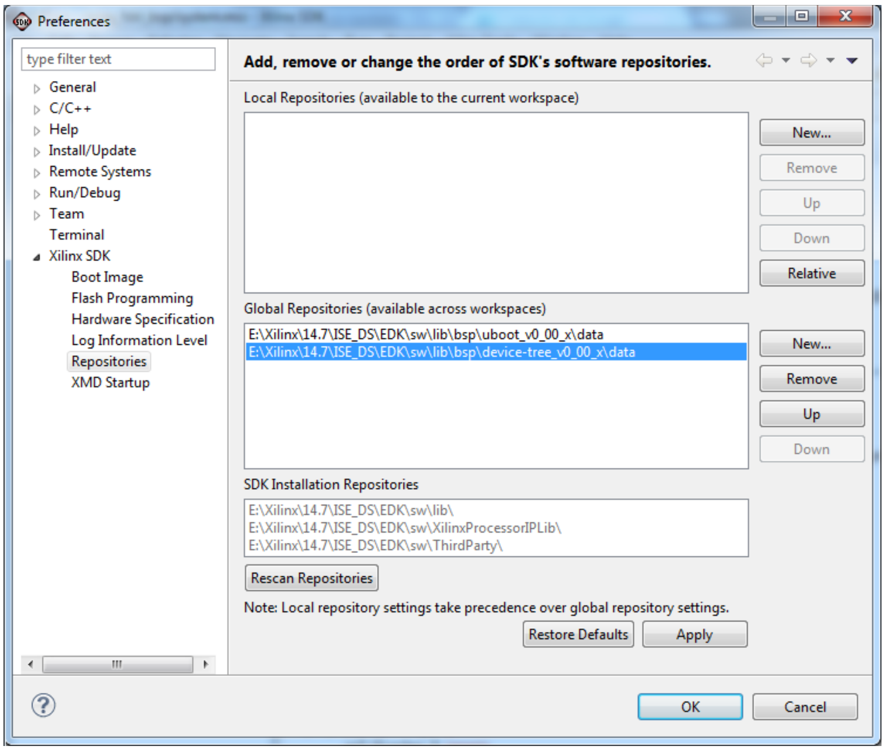
- Download the **Linux Device Tree Generator** from its git repository, add it to Xilinx SDK and build the custom DTS file for a specific hardware (recommended for custom hardware configuration).

- Download the official **Linux Xilinx repository** (linux-xlnx), which already contains the default DTS file for ZedBoard.
- Configure **BuildRoot** to automatically compile the DTB file from the default DTS file for ZedBoard (recommended for default hardware configuration).

Emphasize that the **DTB** and, therefore the DTS, is **closely related** to the **Linux Kernel** file, which is also closely related to the Linux Root File System Image. Therefore, it will be impossible to build these files independently. This is the season why to build this file in the first method will be downloaded the linux-xlnx git directory and why in the in BuildRoot configuration window, the DTB options will be located along with the Linux Kernel options.

In addition, the methodology will be the same in the last 2 methods, in the same way that in the U-Boot case.

The **first method** is the longer way to create the DTS. Nevertheless, it is the only method which is able to create a specific DTS file for any custom hardware. Therefore, it is very important to know it. The Device Tree Generator is a Xilinx EDK tool that plugs into the Automatic BSP Generation features of the tool SDK and produces a Device Tree Source file with the information of the designed hardware which was already configured in PlanAhead. After downloading it, it must be added to Xilinx SDK as a new repository like U-Boot.



Add, remove or change the order of SDK's software repositories.

Local Repositories (available to the current workspace)

Global Repositories (available across workspaces)

E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\bsp\uboot_v0_00_x\data
E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\bsp\device-tree_v0_00_x\data

SDK Installation Repositories

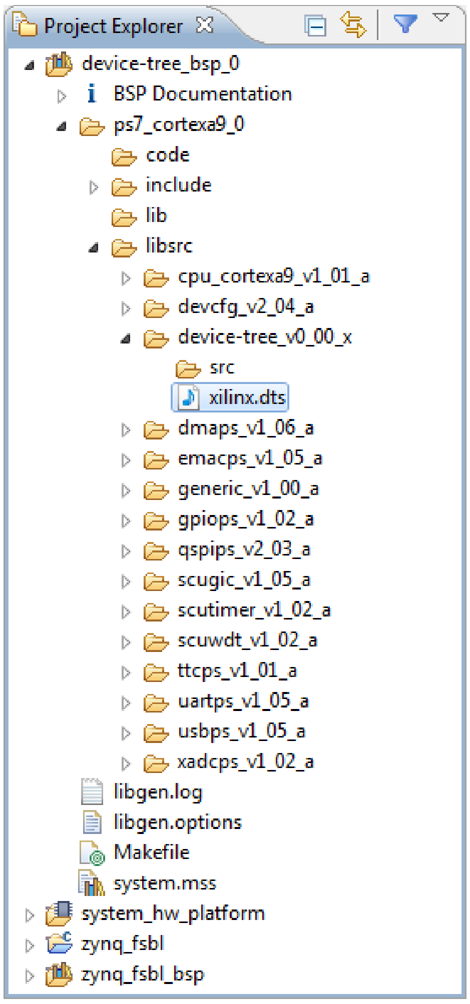
E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\
E:\Xilinx\14.7\ISE_DS\EDK\sw\XilinxProcessorIPLib\
E:\Xilinx\14.7\ISE_DS\EDK\sw\ThirdParty\

Note: Local repository settings take precedence over global repository settings.

Figure 32: Adding a Board Support Package in Xilinx SDK 2/2

After adding the Device Tree Generator, the required DTS file can be created after creating a new **Board Support Package** choosing the “device-tree” option, like U-Boot was made.

Therefore, the created DTS file is customized for the specific hardware selected. Nevertheless, the hardware has a default configuration in this case. Therefore, the following two methods are also valid. The DTS file can be displayed in the SDK “Project Explorer” window.



```
Project Explorer
device-tree_bsp_0
    BSP Documentation
    ps7_cortexa9_0
        code
        include
        lib
        libsrc
            cpu_cortexa9_v1_01_a
            devcfg_v2_04_a
            device-tree_v0_00_x
                src
                xilinx.dts
            dmaps_v1_06_a
            emacps_v1_05_a
            generic_v1_00_a
            gpiops_v1_02_a
            qspips_v2_03_a
            scugic_v1_05_a
            scutimer_v1_02_a
            scuwdt_v1_02_a
            ttcps_v1_01_a
            uartps_v1_05_a
            usbps_v1_05_a
            xadcps_v1_02_a
    libgen.log
    libgen.options
    Makefile
    system.mss
system_hw_platform
zynq_fsbl
zynq_fsbl_bsp
```

Figure 33: DTS File in SDK

The **second method**, downloading the linux-xlnx repository, directly provides the DTS file of almost available boards/architectures. This TDS files can be found in:

.../linux-xlnx/arch/arm/boot/dts/zynq-zed.dts

Likewise, in the **third method**, BuildRoot will download and configure the specified Linux repository (linux-xlnx), so the default file will also be available in the same subfolder:

.../buildroot/output/build/linux-xilinx-v14.5/arch/arm/boot/dts/zynq-zed.dts

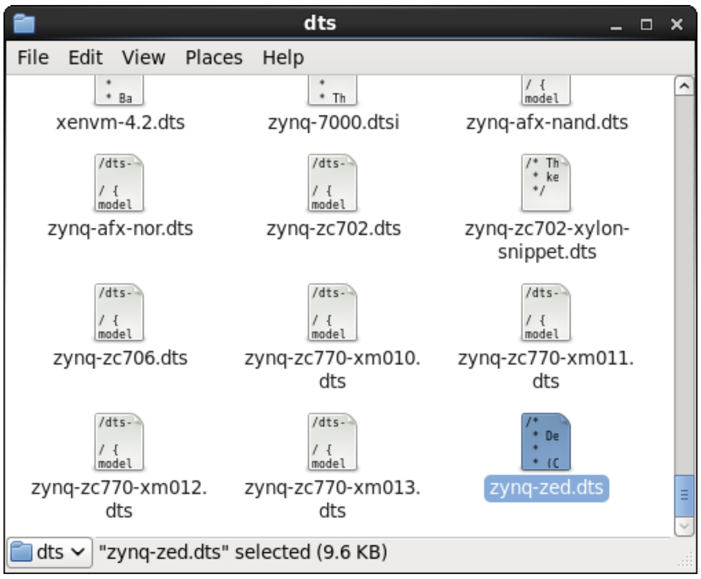


Figure 34: Linux-xlnx Folder with the Default DTS Files

3.7.2. Device Tree Binary (DTB File)

The required DTS file is available. However, it is necessary to convert this human-readable file in a proper binary machine-readable file for U-Boot and Linux to understand it. The Device Tree Compiler (dtc) is responsible for carrying out this work. One again, there are **different options** to create the DTB file from the DTS file.

- Download the **Device Tree Compiler (dtc)** for Linux and build the DTB file.
- Use the dtc available in the **linux-xlxn** folder and create it manually like in the previous method. It is located under scripts/dtc in the Linux kernel source.
- **Do nothing**, when the Kernel Image and/or the Root File System Image are built, the DTB file is automatically created from the pre-indicated DTS source (recommended).

The linux-xlxn folder has its own dtc compiler; therefore, it is completely unnecessary to use the first method because the second method is configured and executed in the same way, since the dtc used in each one is the same. In addition, the third method realizes the same steps than the second method, but it executes them automatically.

Therefore, the **second method** will be shown in order to better understanding how BuildRoot automatically performs this action in the third method.

Once open the Linux "Terminal" command, to convert the DTS file is only required one command line:

```
      <linux-xlnx_path>/linux-xlnx/scripts/dtc/dtc -I dts -O dtb -o
<output_file_path>/devicetree.dtb <input_file_path>/<input_file_name>
```

Command Window 1: Building the DTB File 1/2

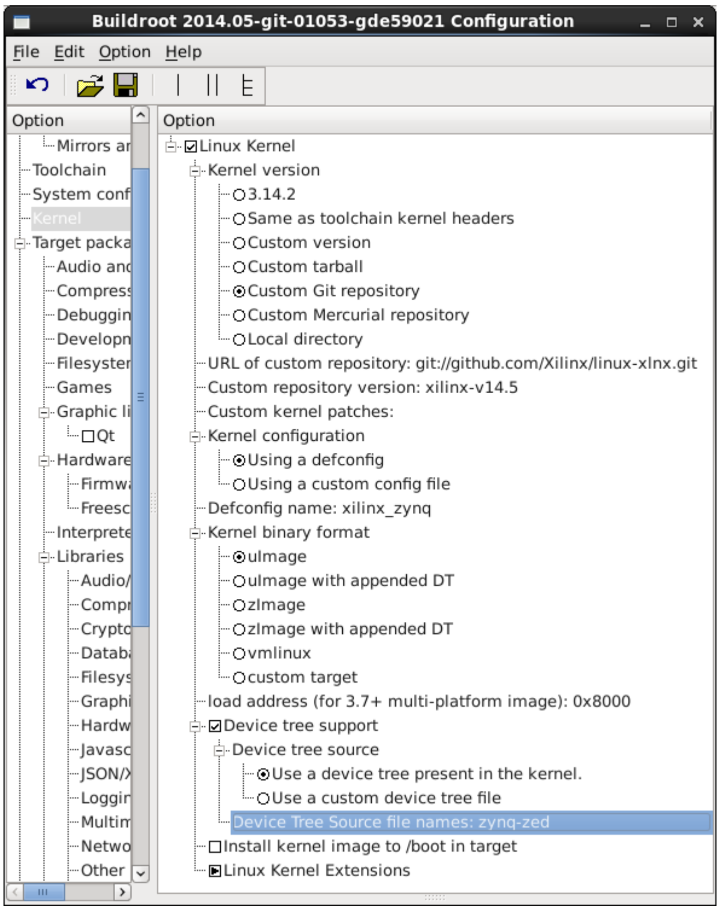
This command line will use the "dtc" file found in "<linux-xlnx_path>/linux-xlnx/scripts/dtc/". Its input argument (-I) is a "dts" file and the output argument (-O) will be a "dtb" file. Moreover, it will take as input the <input_file_name> file which can be located in the <input_file_path> and will return the "devicetree.dtb" file which will be located in <output_file_path>. Note that the inverse process can also be performed.

```
     <linux-xlnx_path>/linux-xlnx/scripts/dtc/dtc  -I  dtb  -O  dtc  -o
<output_file_path>/<output_file_name> <input_file_path>/devicetree.dtb
```

Command Window 2: Building the DTB File 2/2

The **last method** simply consists on allowing to Linux-xlnx to perform this step automatically at the same time in which the Kernel Image and Root File System Image are created. It is only necessary to indicate the source DTS file which will be converted. In the case of using BuildRoot, there are **two options**. On one hand, if there is not a specific custom hardware, indicate as source the **pre-configured DTS file**, like in the following figure. On the other hand, if exists a **specific DTS file**, such as the previous "xilinx.dts", select "Use a custom device tree file" and indicated the file path of this file.

Remember that the "make zedboard_defconfig" command loads the **default configuration** automatically for ZedBoard, and includes this default Device Tree Binary configuration.



```
☑ Linux Kernel
  Kernel version
    ○ 3.14.2
    ○ Same as toolchain kernel headers
    ○ Custom version
    ○ Custom tarball
    ● Custom Git repository
    ○ Custom Mercurial repository
    ○ Local directory
  URL of custom repository: git://github.com/Xilinx/linux-xlnx.git
  Custom repository version: xilinx-v14.5
  Custom kernel patches:
  Kernel configuration
    ● Using a defconfig
    ○ Using a custom config file
  Defconfig name: xilinx_zynq
  Kernel binary format
    ● uImage
    ○ uImage with appended DT
    ○ zImage
    ○ zImage with appended DT
    ○ vmlinux
    ○ custom target
  load address (for 3.7+ multi-platform image): 0x8000
  ☑ Device tree support
    Device tree source
      ● Use a device tree present in the kernel.
      ○ Use a custom device tree file
    Device Tree Source file names: zynq-zed
  ☐ Install kernel image to /boot in target
  Linux Kernel Extensions
```

Figure 35: Creating the "devicetree.dtb" File in BuildRoot

Note that the **Device Tree** support settings in BuildRoot are in the **same window** that the **Kernel** configuration, because the Linux Kernel needs it to know the machine hardware configuration. Therefore, the specific Operating System which is going to be implemented depends on the Device Tree Binary.

3.8. Linux Kernel File, “uImage”

So far, some **Linux Kernel** features have been mentioned, but “what the Linux Kernel is” has not been explained. Therefore, it is time to know what the Linux Kernel is and which responsibilities it has to realize exactly. A kernel is the lowest level of easily replaceable software that interfaces with the hardware in the device. It is responsible for interfacing all of the applications which are running in “user mode” down to the physical hardware, and allowing processes, known as servers, to get information from each other using inter-process communication (IPC).

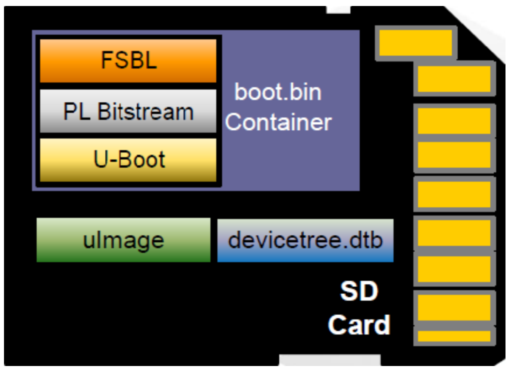


Figure 36: The “uImage” File

There are, of course, different ways to build a kernel and architectural considerations when building one from scratch. In general, most kernels fall into one of three types: monolithic, microkernel, and hybrid. Specifically, the Linux Kernel is a monolithic kernel. **Monolithic kernels** are the opposite of Microkernels because they encompass not only the CPU, memory, and IPC, but they also include device drivers, file system management, and system server calls. Monolithic kernels tend to be better at accessing hardware and multitasking because if a program needs to get information from memory or another process running it has a more direct line to access it and does not have to wait in a queue. Nevertheless, this can cause problems because the more things which run in supervisor mode, the more things which can bring down the system. Therefore, there are **advantages and disadvantages** by using these kernels:

+ More direct access to hardware for programs.
+ Easier for processes to communicate between each other.
+ If a device is supported, it should work with no additional installations.
+ Processes react faster because there is not a queue for processor time.
– Large install footprint.
– Large memory footprint.
– Less secure because everything runs in supervisor mode.

Therefore, because the Linux Kernel is monolithic, it has the **largest footprint** and the most complexity over the other types of kernels. This was a design feature which was under quite a bit of debate in the early days of Linux and still carries some of the same design flaws that monolithic kernels are inherent to have.

In order to avoid these flaws, the **kernel** modules can be **loaded and unloaded at runtime**, meaning the **features** of the kernel can be **added or removed on the fly**. This can go beyond just adding hardware functionality to the kernel, by including modules that run server processes, but it can also allow the entire kernel (or at least most of it) to be replaced without needing to reboot the device.

In addition, the Linux Kernel initializes the system hardware and mounts the Root File System Image. It is highly configurable and allows adding/removing **support** for:

- Debugging.
- Specific device drivers.
- Types of file systems.
- Boot options.

Once again, there are **two different methods** to build the Linux Kernel file. On one hand, it can be build using the linux-xlnx directory. On the other hand, BuildRoot can be configured and used to build the kernel. It will be showed how the linux-xlnx directory builds the Kernel file in order to better understanding how BuildRoot performs this task automatically.

Using the **first method**, there is a default configuration file available in the linux-xlnx folder to prepare the Linux Kernel for ZedBoard: "xilinx_zynq_defconfig". It is located in the following directory path:

…/linux-xlnx/arch/arm/configs/xilinx_zynq_defconfig

Therefore, the Linux Kernel file has to be configured for ZedBoard before building it:

```
make ARCH=arm CROSS_COMPILE=arm-linux- xilinx_zynq_defconfig
```

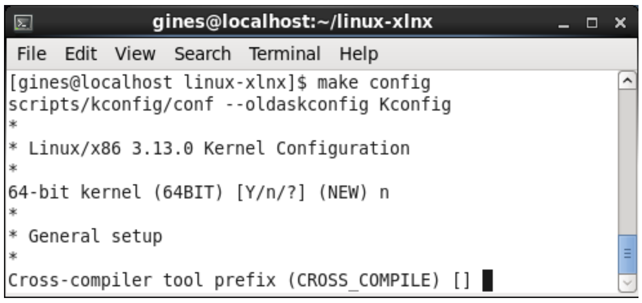
Command Window 3: Setting the Linux Kernel File for ZedBoard

This command prepares the kernel source tree for the ZedBoard including some special configuration by setting the environment variables:

- **"ARCH="** sets the arquitecture, ARM in this case.
- **"CROSS_COMPILE="** sets the cross tolchain, arm-xilinx-linux-gnueabi- in this case, which will use the xilinx_zynq_defconfig configuration file.

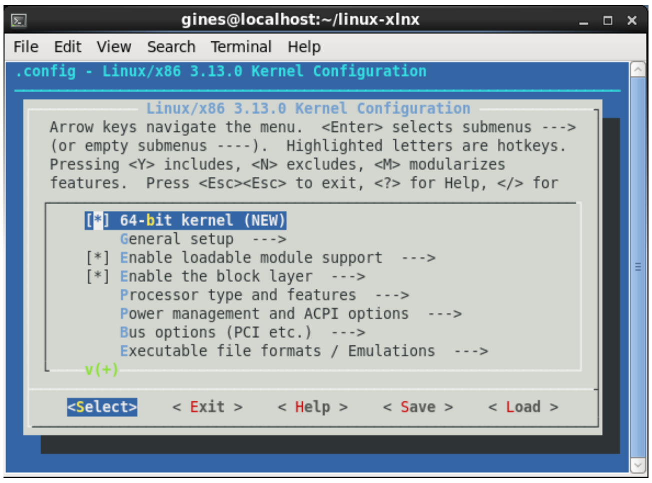
As result, a default configuration ".config" file is built in the linux-xlnx folder. In order to deeply configure and customize the Linux Kernel, different configuration editors can be used by typing the followings commands.

- **“make config”**: configure the “.config” file using a line-oriented program.
- **“make menuconfig”**: configure the “.config” file using a menu-based program.
- **“make xconfig”**: configure the “.config” file using a QT-based front end.
- **“make gconfig”**: configure the “.config” file using a GTK-based front end.


gines@localhost:~/linux-xlnx
File Edit View Search Terminal Help

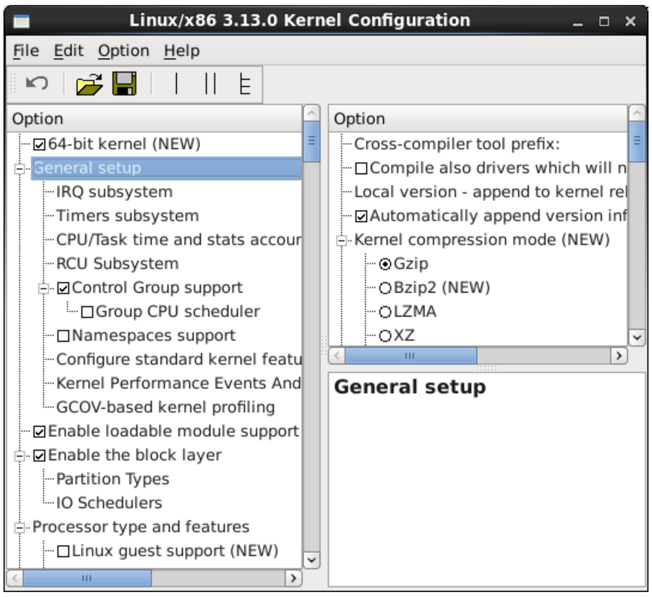
```
[gines@localhost linux-xlnx]$ make config
scripts/kconfig/conf --oldaskconfig Kconfig
*
* Linux/x86 3.13.0 Kernel Configuration
*
64-bit kernel (64BIT) [Y/n/?] (NEW) n
*
* General setup
*
Cross-compiler tool prefix (CROSS_COMPILE) []
```

Figure 37: “config” Configuration Editor


gines@localhost:~/linux-xlnx
File Edit View Search Terminal Help

```
.config - Linux/x86 3.13.0 Kernel Configuration
                Linux/x86 3.13.0 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->
  (or empty submenus ----).  Highlighted letters are hotkeys.
  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for

      [*] 64-bit kernel (NEW)
          General setup  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          Processor type and features  --->
          Power management and ACPI options  --->
          Bus options (PCI etc.)  --->
          Executable file formats / Emulations  --->
      v(+)

     <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 38: “menuconfig” Configuration Editor



```
☑ 64-bit kernel (NEW)
General setup
    IRQ subsystem
    Timers subsystem
    CPU/Task time and stats accour
    RCU Subsystem
    ☑ Control Group support
        ☐ Group CPU scheduler
    ☐ Namespaces support
    Configure standard kernel featu
    Kernel Performance Events And
    GCOV-based kernel profiling
☑ Enable loadable module support
☑ Enable the block layer
    Partition Types
    IO Schedulers
Processor type and features
    ☐ Linux guest support (NEW)
```

```
Cross-compiler tool prefix:
☐ Compile also drivers which will n
Local version - append to kernel rel
☑ Automatically append version inf
Kernel compression mode (NEW)
    ⦿ Gzip
    ○ Bzip2 (NEW)
    ○ LZMA
    ○ XZ
```

General setup

Figure 39: “xconfig” Configuration Editor

Note that the four configuration **editors** performs the **same configuration**, it does not matter the chosen editor. Once the Kernel has been customized and prepared, it can be built with the "make" command and it will appear in the following directory.

.../linux-xilinx/arch/arm/boot/uImage

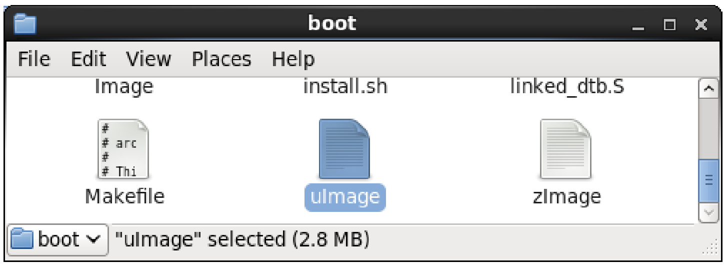
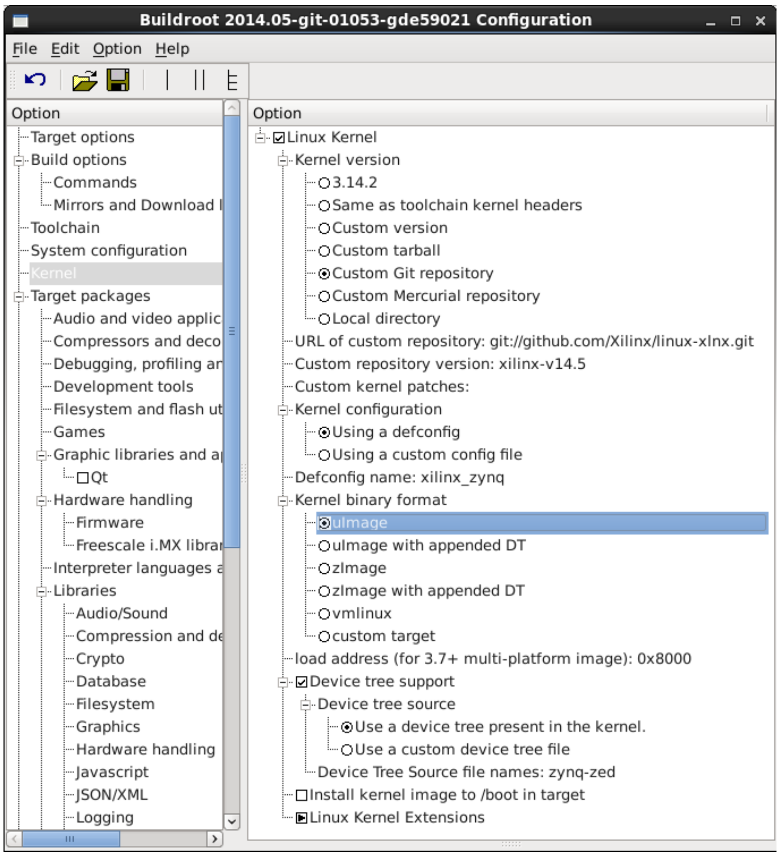


Figure 40: "uImage" Location Folder

Likewise, in the **second method**, BuildRoot will download and pre-configure the specified Linux repository (linux-xlnx), so only the specific configuration has to be done. It can be done by typing any previous configs commands, such as the "make xconfig":



```
Option
[✓] Linux Kernel
    Kernel version
        ○ 3.14.2
        ○ Same as toolchain kernel headers
        ○ Custom version
        ○ Custom tarball
        ● Custom Git repository
        ○ Custom Mercurial repository
        ○ Local directory
    URL of custom repository: git://github.com/Xilinx/linux-xlnx.git
    Custom repository version: xilinx-v14.5
    Custom kernel patches:
    Kernel configuration
        ● Using a defconfig
        ○ Using a custom config file
    Defconfig name: xilinx_zynq
    Kernel binary format
        ● uImage
        ○ uImage with appended DT
        ○ zImage
        ○ zImage with appended DT
        ○ vmlinux
        ○ custom target
    load address (for 3.7+ multi-platform image): 0x8000
    [✓] Device tree support
        Device tree source
            ● Use a device tree present in the kernel.
            ○ Use a custom device tree file
        Device Tree Source file names: zynq-zed
    [ ] Install kernel image to /boot in target
    [■] Linux Kernel Extensions
```

Figure 41: Configuring the Linux Kernel in BuildRoot

Remember that the "make zedboard_defconfig" command loads the **default configuration** automatically for ZedBoard, and includes this default Linux Kernel configuration. In addition, the "uImage" file is generated in the same folder than in the previous method.

Nevertheless, BuildRoot also copies it in the following folder, as had already occurred with the U-Boot file and the Device Tree Binary file:

.../buildroot/output/images/uImage

Otherwise, note that the "uImage" will be built. Nevertheless, **linux-xlnx generates** two more files, "Image" and "zImage".

- **Image** is the generic Linux kernel binary image file.
- **zImage** is a compressed version of the Linux kernel image that is self-extracting.
- **uImage** is a zImage file that has a U-Boot wrapper (installed by the mkimage utility) that includes the OS type and loader information. Since this file is self-extracting (i.e. needs no external decompressors), the wrapper will indicate that this kernel is "not compressed" even though it actually is.

Any of them could be used as **Linux Kernel** file. Nevertheless, for economy of storage, a compressed image will be more convenient. In addition, "zImage" files are not compatible with U-Boot, it must be converted into a "uImage" file. Because of this, the "uImage" file will be chosen.

Finally, a **summary** of the Linux Kernel build process in BuildRoot is shown below:

1. Selecting the Linux Kernel version.
2. Loading the xilinx_zynq_defconfig configuration file for ZedBoard.
3. Specifying the Kernel Binary Format.
4. Configuring the Linux Kernel by launching the Linux configuration window by the "make linux-menuconfig" command.
5. Checking/unchecking extra options in the "menuconfig" command.
6. Setting the Kernel Build & Cross Compilation.
7. Choosing the Image Compression ("uImage").
8. Generating the "uImage" file.

Note that **not** all of these steps are **mandatory**, the default settings could be used and some of them may be skipped. It will depend on the programmer requirements.

3.9. Root File System Image, "uramdisk.image.gz"

Finally, three of the four required files are built. It is only necessary one more file, the **Root File System Image**. It is a compressed file which contains all operating system files.

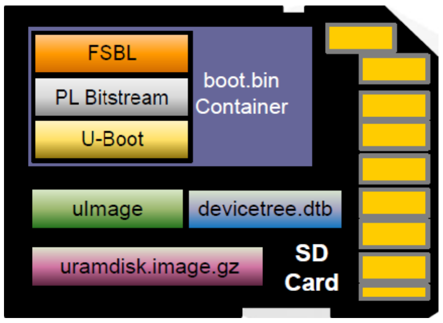


Figure 42: The "uramdisk.image.gz" File

It is **composed of different folders and files**, the same ones that a normal Linux OS installed in any personal computer or laptop has. Nevertheless, it contains only the necessary folders which are going to be used for the specific application for which has been designed. The most important **folders** which usually will appear in any custom and embedded Linux OS are explained below:

- **"dev"** contains the device drivers (to connect peripherals).
- **"lib"** and **"lib32"** contain the system libraries for 64 and 32 bits respectively (similar to "C:\Windows\System" in Windows).
- **"mnt"** is the mount point for other file systems. Linux only allows one root file system but other disk can be added by mounting them to a directory in the root file system. It is similar to mapping a drive under Windows.
- **"root"** and **"home"** are the storage folders for super user files and for the rest of users respectively (similar to "My Documents" in Windows).
- **"sys"** and **"proc"** are the virtual file systems location, and expose the kernel parameters as files (similar to Windows Registry).
- **"usr"** is the storage folder for user binaries, all Linux System programs are stored in here (similar to "Program Files" in Windows).

In addition, there is also an important file, **"init"**. It is the first userspace program started by the kernel, and is responsible for starting the userspace services and programs.

As it was the Linux Kernel building, the Root File System Image can be configured in the linux-xlnx folder or using BuildRoot. It will be directly built using BuildRoot because of its greater simplicity, as has already made earlier.

A **summary** of the Root File System Image build process is shown below:

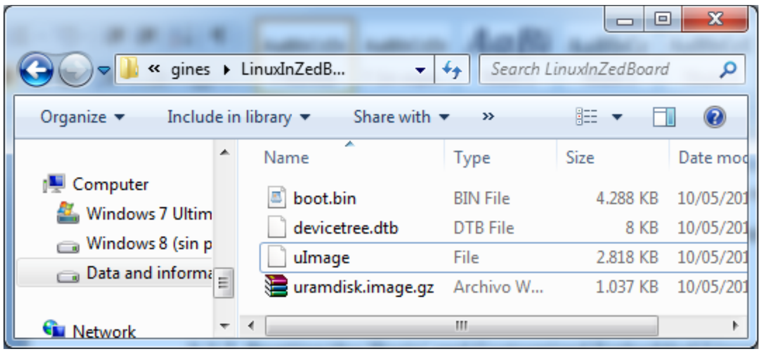
1. **BuildRoot** configuration ("make xconfig" command):
    a. Setting the Architecture and CPU.
    b. Specifying the Toolchain & Cross Compiler version.
    c. Setting the Kernel Headers.
    d. Specifying the µClibC library version.
    e. Specifying the Binutils version.
    f. Specifying the Cross Compiler version.
    g. Specifying the repositories of the Linux Sources, such as Busybox, µClibC, etc.
2. **Busybox** configuration ("make busybox-menuconfig" command in BuildRoot):
    a. Adding utilities and User Space Applications.
3. **µClibC** configuration ("make udibc-menuconfig" command):
    a. Configuring the Library Settings for the Target.
4. **Build and Cross Compilation**.
5. **"ramdisk.image.gz"** file generation.
6. **"uramdisk.image.gz"** file which is obtained from the "ramdisk.image.gz" file.

Note that not all of these steps are mandatory, the default settings could be used and some of them may be skipped. It will depend on the programmer requirements.

In addition, all these steps will be showed (and briefly explained if it is necessary) in "Appendix 1: Guide - Linux on ZedBoard step by step".

3.10. Booting the Custom Embedded Linux OS

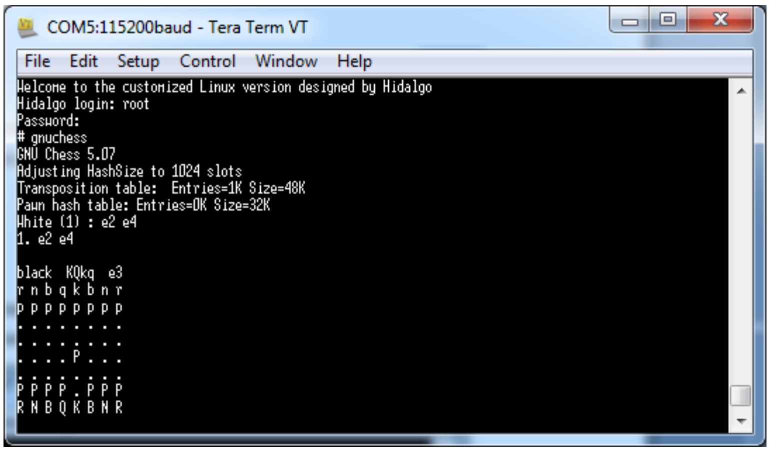
Once the Root File System Image is built, the **four required files** which the SD Card needs are **available**. Therefore, the "Basic" Embedded Linux OS can be booted on ZedBoard.



Name	Type	Size	Date mod
boot.bin	BIN File	4.288 KB	10/05/201
devicetree.dtb	DTB File	8 KB	10/05/201
uImage	File	2.818 KB	10/05/201
uramdisk.image.gz	Archivo W...	1.037 KB	10/05/201

Figure 43: SD Card Required Files

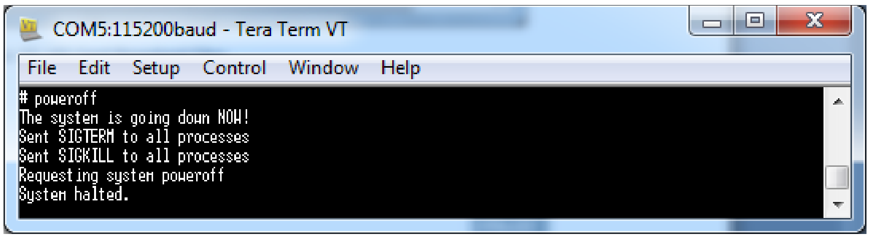
It contains a chess game in order to be able to test if Linux works properly, which can be launched by typing "**gnuchess**" and display with the Tera Term program in any computer or laptop. The user name and password in this case are "root".


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
Welcome to the customized Linux version designed by Hidalgo
Hidalgo login: root
Password:
# gnuchess
GNU Chess 5.07
Adjusting HashSize to 1024 slots
Transposition table:  Entries=1K Size=48K
Pawn hash table: Entries=0K Size=32K
White (1) : e2 e4
1. e2 e4

black  KQkq  e3
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . P . . .
. . . . . . . .
P P P P . P P P
R N B Q K B N R
```

Figure 44: GNUChess Displaying

The "**poweroff**" command can be typed to shut down the Operating System. Once Linux is switched off, the ZedBoard can be turned off too.


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
# poweroff
The system is going down NOW!
Sent SIGTERM to all processes
Sent SIGKILL to all processes
Requesting system poweroff
System halted.
```

Figure 45: Shutting Down the Operating System

Finally, note that Buildroot provides several ways of **extra customization** if a more custom configuration is required.

One important configuration method is the **filesystem customization**. As already mentioned, the target filesystem is available in the folder showed below. So if the developer wants to add or delete folders and runs the "make" command afterwards, the target filesystem image will be rebuilt.

…/buildroot/output/target/

Nevertheless, this directory does not contain the root filesystem which will be used on ZedBoard. Since BuildRoot does not run as root, it cannot create device files and set the permissions and ownership of files correctly in this directory to make it usable as a root filesystem.

For that reason, the contents of **this directory cannot** be used to **mount** the root filesystem and cannot be copied **directly** to the **SD Card**. In order to convert these files into a usable root filesystem, the "make" command must be re-executed to compile the changes in this skeleton and re-build the filesystem images file in the pre-selected compress format.

Other way is to create a custom **target skeleton**, starting with the default skeleton model available in the folder showed below and then customizing it to fit the needs.

…/buildroot/system/skeleton/

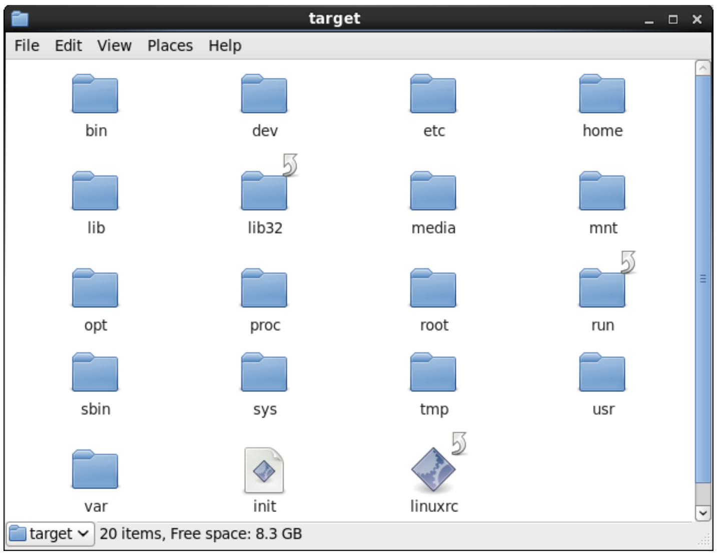


Figure 46: File System Files

Finally, note that other **extra configurations** can be realized by configuring the downloaded BuildRoot directories:

- Extra **µClibc** configuration by typing "make udibc-menuconfig".
- Extra **Linux-xlnx** configuration by typing "make linux-menuconfig".
- Extra **BusyBox** configuration by typing "make busybox-menuconfig".
- Extra **BareBox** configuration by typing "make barebox-menuconfig".

Chapter 4: Ubuntu Linux OS on ZedBoard

4.1. Overview

In this section, the aim is to implement a **GNU pre-designed OS** in which the specific PL peripheral will be easily added. This option, unlike the "Basic" Linux OS developed previously, has different but also interesting advantages. In this case, the chosen pre-designed OS is **Linaro-Ubuntu Linux OS**, because it has already pre-configured the drivers to connect a HDMI monitor, a keyboard and a mouse to the ZedBoard.

A Linux kernel is used as the foundation operating system running on the processor cores, but also is added a fully featured desktop from Ubuntu. The desktop allows the **ZedBoard** to function **as** a **personal computer** using a USB Keyboard and mouse, along with an HDMI monitor. With Linaro-Ubuntu installed on ZedBoard, a vast array of applications can be installed and used, just as if the ZedBoard was a normal PC. This includes the potential to develop in a native application development environment on the ARM system.

The **reason** to develop a desktop OS in an embedded system is that is the future of some technology, such as mobile computing. Some Smartphones are commercially available with an Ubuntu desktop and an Android environment running side by side on top of a Linux kernel. This is possible because the Ubuntu desktop and the Java Virtual Machine use a common Linux kernel.

If the documentation of the Linaro-Ubuntu Linux is read, it can be seen that the same first three **files** are **required** in the SD Card to start this Linux OS on ZedBoard, which will be saved in a FAT32 SD Card partition, such as with the previous Linux OS. Nevertheless, the Root File System Image will not be stored in this partition, but it will be saved in an ext4 partition.

1. **FAT32** partition:
    a. The **Boot Image** ("boot.bin"), which requires the same three components:
        a. The **FSBL** (First Stage Boot Loader).
        b. The **Programmable Logic Hardware BitStream** (optional).
        c. The **U-Boot** (Second Stage Boot Loader).
    b. The **Device Tree Binary File** ("devicetree.dtb"), which is again obtained from the Device Tree Source File and loaded into the DDR memory by U-Boot.
    c. The **Linux Kernel File** ("uImage"), which is again loaded into the DDR memory by U-Boot.
2. **ext4** partition:
    a. The **Root File System Image**, which is also loaded into the DDR memory by U-Boot. It contains the Operating System itself. Nevertheless, it will not be a compress file in this case.

Again, the next figure can summarize these steps, showing the typical Linux Boot sequence.

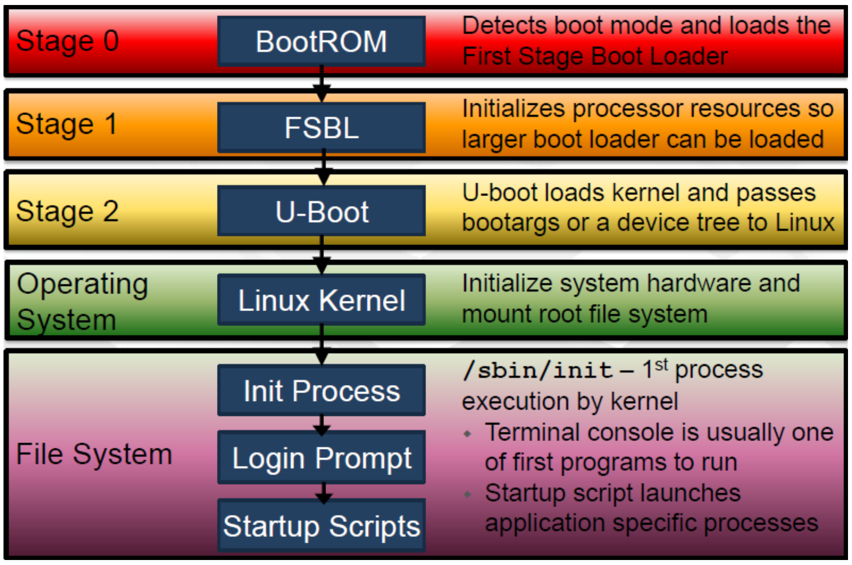

Stage 0
BootROM
Detects boot mode and loads the First Stage Boot Loader
Stage 1
FSBL
Initializes processor resources so larger boot loader can be loaded
Stage 2
U-Boot
U-boot loads kernel and passes bootargs or a device tree to Linux
Operating System
Linux Kernel
Initialize system hardware and mount root file system
File System
Init Process
Login Prompt
Startup Scripts
/sbin/init – 1st process execution by kernel
Terminal console is usually one of first programs to run
Startup script launches application specific processes

Figure 47: Typical Linux Boot Sequence Overview

All these files will be explained in further detail in this chapter. Remember that the order in which these files will be made is different to the **Linux Boot Sequence** (e.g. the BitStream is the first required file which is created).

Before performing this task, a **new project could be created** and configured for ZedBoard in PlanAhead, in order to be able to create a specific FSBL and BitStream files for the specific hardware, the ZedBoard. In addition, the HDMI monitor, the mouse and the keyboard have to be configured.

Nevertheless, **one of the main advantages** of this kind of OS is that pre-configured projects can be found on Internet. In this way, it is not necessary to waste time in configured the whole set of peripherals when other people have already configured them. If some extra-configuration has to be done, it will be performed from this pre-designed project (e.g. add or drop peripherals).

Again, **Xilinx** and its whole family of programs will be the main tool for achieving this goal, **together** with some **GNU tools**. Therefore, the programs and steps to create, configure, build and implement a Linaro-Ubuntu Linux OS on ZedBoard are summarized below.

1. **VMware Player:**
   a. Preparing the SD Card.
2. **Xilinx Platform Studio (XPS):**
   a. Downloading and loading the Project.
   b. Generating the BitStream File and exporting to SDK.
3. **Software Development Kit (SDK):**
   a. Creating the FSBL (First Stage Boot Loader).
4. **VMware Player:**
   a. Configuring and obtaining the U-Boot file.

5. **Software Development Kit (SDK):**
    a. Creating the boot image ("boot.bin") from the FSBL, the BitStream file and the U-Boot file.
6. **VMware Player:**
    a. Configuring and obtaining the Device Tree Binary file.
    b. Configuring and obtaining the Kernel Image.
7. **SD Card:**
    a. Copying the files in the SD Card.
8. **Tera Term:**
    a. Running Linaro-Ubuntu Linux on ZedBoard.

Note that **some steps** will be **reuse from "Chapter 3**: Custom Embedded Linux OS on ZedBoard", because of the necessity of almost the same files. Nevertheless, these files will have different configuration.

Finally, remember that this chapter is **closely related** to "Appendix 1: Guide – Linux on ZedBoard step by step"; they both are complementary to each other. This section will be focus on the idea, on the concepts, on answering the questions "what must be do", "why must be do" and "which ways are there available" to develop and implement the Linux OS. Nevertheless, the appendix will be focused on the exactly steps to achieve the target, on answering "how must be exactly done to develop and implement it".

Therefore, this appendix is strongly recommended after reading this chapter in order to better understand the ideas exposed in this chapter.

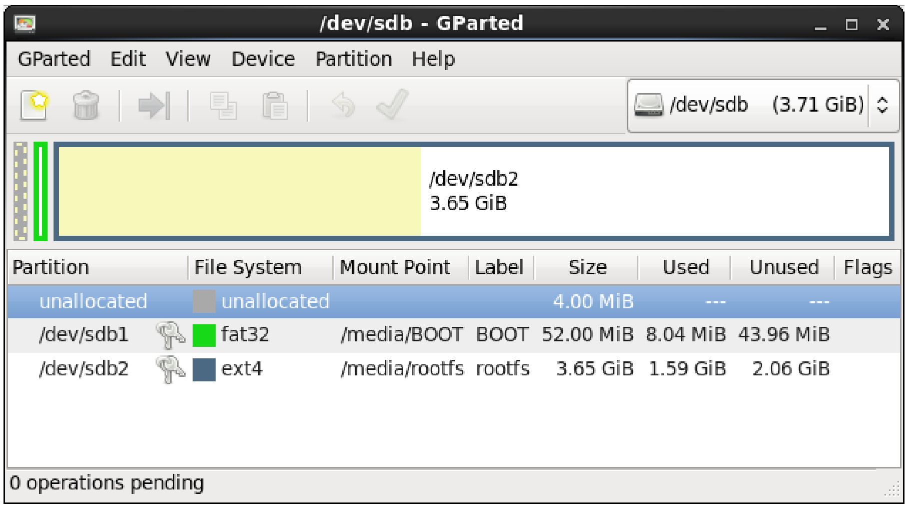
4.2. Preparing the SD Card

Again, the ZedBoard will be booted from data contained on the SD Card. Nevertheless, this Linux OS has a very huge size, being **not practical** to use a **compress** "uRamDisk.image.gz" **file** because it will require wasting a lot of time in decompress the file every time that the ZedBoard is switched on. As a solution, the Root File System Image will reside in another partition of the SD Card, without being compressed.

Therefore, it must be created **two partitions** on the SD card. The first one will be in **FAT32** format, visible and accessible by either a Windows or Linux OS. This partition will contain the files used for initial boot of the ZedBoard (i.e. FSBL file, Device Tree file and Linux Kernel file). The second one will be in **ext4** format, readable and writable only by a Linux OS. On this partition, the Root File System will be placed. In this case, the first FAT32 format partition will have 52MB and the second one will use the remaining space in ext4 format.

These partitions will be prepared in VMware, because ext4 is only visible for a Linux OS. The chosen tool is **GParted Partition Editor** (Gnome Partition Utility). GParted is a GNU program to create, modify and remove disk partitions.

The first step is to delete any previous partitions in the SD Card; the second step is to create a 52MB FAT32 partition called "BOOT"; and finally, to create an ext4 partition with the remaining space called "rootfs". Finally, note that if the SD Card is open with Windows, only the 52MB partition will be displayed, because Windows cannot read the ext4 format.



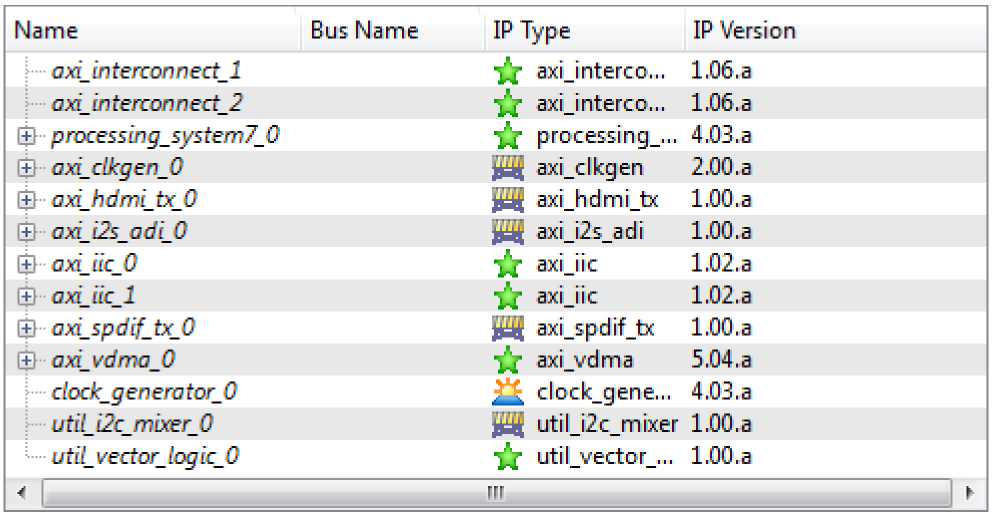
Partition	File System	Mount Point	Label	Size	Used	Unused	Flags
unallocated	unallocated			4.00 MiB	---	---	
/dev/sdb1	fat32	/media/BOOT	BOOT	52.00 MiB	8.04 MiB	43.96 MiB	
/dev/sdb2	ext4	/media/rootfs	rootfs	3.65 GiB	1.59 GiB	2.06 GiB	

0 operations pending

Figure 48: SD Card Prepared for Linaro-Ubuntu

4.3. Downloading and Loading the Project

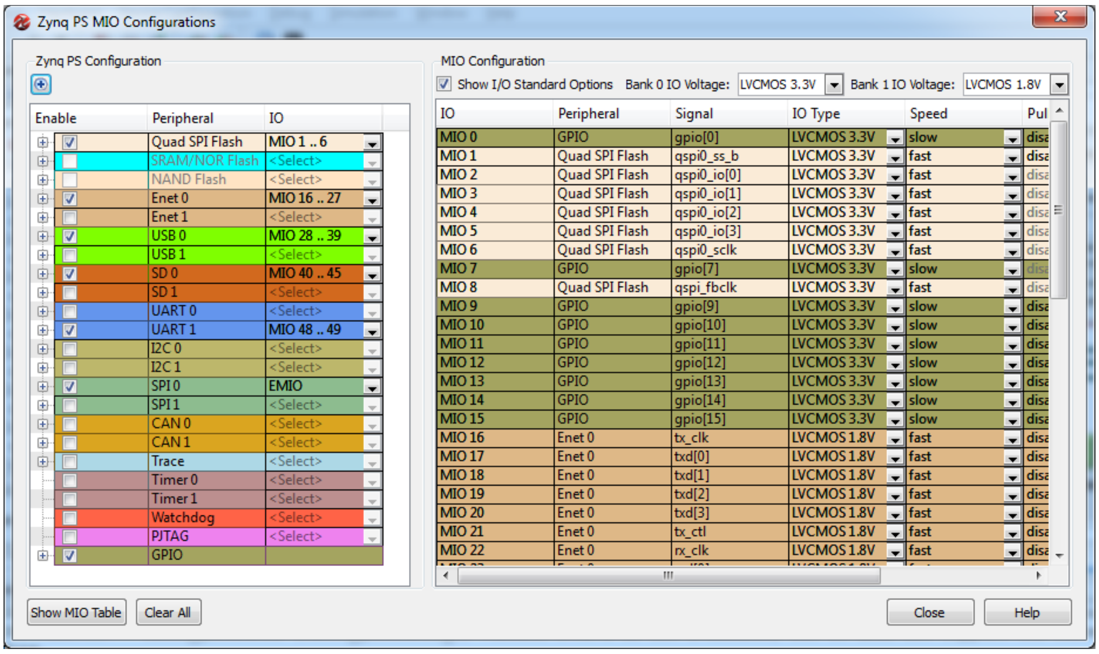
Once the required zip files are downloaded (see Appendix 1), the "system.xmp" file can be opened with XPS. As it was mentioned in the previous chapter, this file contains the **ZedBoard hardware configuration**. For instance, the **HDMI and USB interfaces** (for the monitor, keyboard and mouse), among others, are pre-configured. They can be showed in the "Bus Interfaces", "Ports" or "Addresses" panels. Additionally, if some extra-hardware configuration would be required (e.g. add or drop peripherals), it can be done thought this file. Remember that in the previous chapter, only the processing system was added.



Name	Bus Name	IP Type	IP Version
axi_interconnect_1		axi_interco...	1.06.a
axi_interconnect_2		axi_interco...	1.06.a
processing_system7_0		processing_...	4.03.a
axi_clkgen_0		axi_clkgen	2.00.a
axi_hdmi_tx_0		axi_hdmi_tx	1.00.a
axi_i2s_adi_0		axi_i2s_adi	1.00.a
axi_iic_0		axi_iic	1.02.a
axi_iic_1		axi_iic	1.02.a
axi_spdif_tx_0		axi_spdif_tx	1.00.a
axi_vdma_0		axi_vdma	5.04.a
clock_generator_0		clock_gene...	4.03.a
util_i2c_mixer_0		util_i2c_mixer	1.00.a
util_vector_logic_0		util_vector_...	1.00.a

Figure 49: Bus Interfaces Panel

Again, many **peripherals** are enabled in the Processing System with some MIO pins assigned to them in coordination with the ZedBoard layout. Remember that the peripherals are not listed in alphabetical order, but also they are listed in order of priority; despite the MIOs numbers are listed in number order.


Zynq PS MIO Configurations

Zynq PS Configuration

Enable	Peripheral	IO
✓	Quad SPI Flash	MIO 1 .. 6
	SRAM/NOR Flash	<Select>
	NAND Flash	<Select>
✓	Enet 0	MIO 16 .. 27
	Enet 1	<Select>
✓	USB 0	MIO 28 .. 39
	USB 1	<Select>
✓	SD 0	MIO 40 .. 45
	SD 1	<Select>
	UART 0	<Select>
✓	UART 1	MIO 48 .. 49
	I2C 0	<Select>
	I2C 1	<Select>
✓	SPI 0	EMIO
	SPI 1	<Select>
	CAN 0	<Select>
	CAN 1	<Select>
	Trace	<Select>
	Timer 0	<Select>
	Timer 1	<Select>
	Watchdog	<Select>
	PJTAG	<Select>
✓	GPIO	

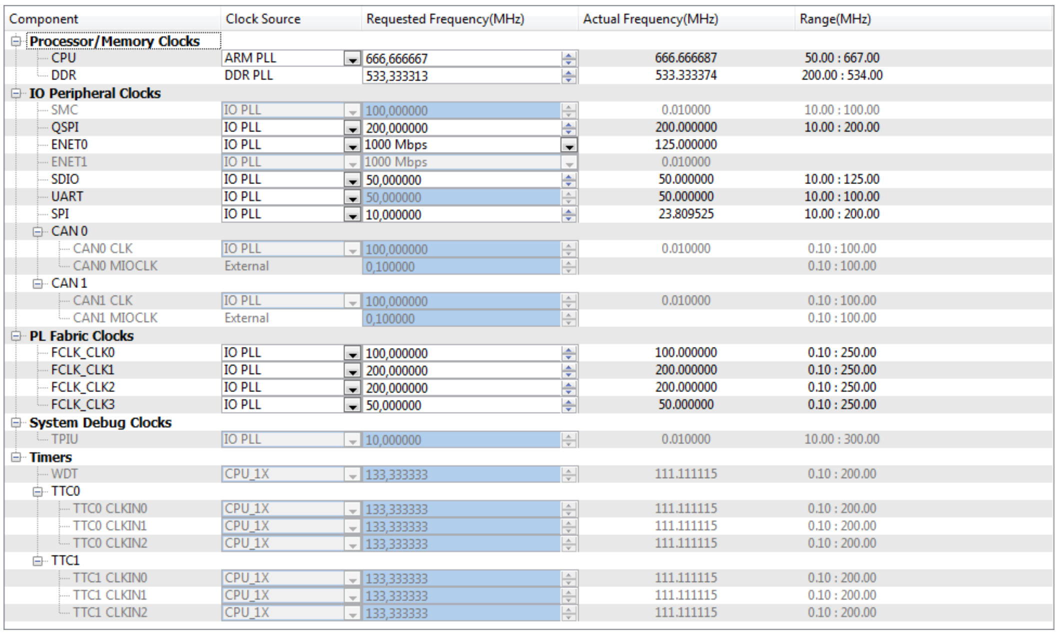
MIO Configuration

✓ Show I/O Standard Options Bank 0 IO Voltage: LVCMOS 3.3V Bank 1 IO Voltage: LVCMOS 1.8V

IO	Peripheral	Signal	IO Type	Speed	Pul
MIO 0	GPIO	gpio[0]	LVCMOS 3.3V	slow	disa
MIO 1	Quad SPI Flash	qspi0_ss_b	LVCMOS 3.3V	fast	disa
MIO 2	Quad SPI Flash	qspi0_io[0]	LVCMOS 3.3V	fast	disa
MIO 3	Quad SPI Flash	qspi0_io[1]	LVCMOS 3.3V	fast	disa
MIO 4	Quad SPI Flash	qspi0_io[2]	LVCMOS 3.3V	fast	disa
MIO 5	Quad SPI Flash	qspi0_io[3]	LVCMOS 3.3V	fast	disa
MIO 6	Quad SPI Flash	qspi0_sclk	LVCMOS 3.3V	fast	disa
MIO 7	GPIO	gpio[7]	LVCMOS 3.3V	slow	disa
MIO 8	Quad SPI Flash	qspi_fbclk	LVCMOS 3.3V	fast	disa
MIO 9	GPIO	gpio[9]	LVCMOS 3.3V	slow	disa
MIO 10	GPIO	gpio[10]	LVCMOS 3.3V	slow	disa
MIO 11	GPIO	gpio[11]	LVCMOS 3.3V	slow	disa
MIO 12	GPIO	gpio[12]	LVCMOS 3.3V	slow	disa
MIO 13	GPIO	gpio[13]	LVCMOS 3.3V	slow	disa
MIO 14	GPIO	gpio[14]	LVCMOS 3.3V	slow	disa
MIO 15	GPIO	gpio[15]	LVCMOS 3.3V	slow	disa
MIO 16	Enet 0	tx_clk	LVCMOS 1.8V	fast	disa
MIO 17	Enet 0	txd[0]	LVCMOS 1.8V	fast	disa
MIO 18	Enet 0	txd[1]	LVCMOS 1.8V	fast	disa
MIO 19	Enet 0	txd[2]	LVCMOS 1.8V	fast	disa
MIO 20	Enet 0	txd[3]	LVCMOS 1.8V	fast	disa
MIO 21	Enet 0	tx_ctl	LVCMOS 1.8V	fast	disa
MIO 22	Enet 0	rx_clk	LVCMOS 1.8V	fast	disa

Figure 50: Zynq PS MIO Configurations

The DDR3 memory has exactly the same configuration as in the previous chapter, because there is only one DDR memory in ZedBoard, so the configuration cannot change among different ZedBoard projects. Nevertheless, the PS PLL Clocks have almost the same configuration; the only difference is that there are new clocks configured.



Component	Clock Source	Requested Frequency(MHz)	Actual Frequency(MHz)	Range(MHz)
Processor/Memory Clocks				
CPU	ARM PLL	666,666667	666.666687	50.00 : 667.00
DDR	DDR PLL	533,333313	533.333374	200.00 : 534.00
IO Peripheral Clocks				
SMC	IO PLL	100,000000	0.010000	10.00 : 100.00
QSPI	IO PLL	200,000000	200.000000	10.00 : 200.00
ENET0	IO PLL	1000 Mbps	125.000000	
ENET1	IO PLL	1000 Mbps	0.010000	
SDIO	IO PLL	50,000000	50.000000	10.00 : 125.00
UART	IO PLL	50,000000	50.000000	10.00 : 100.00
SPI	IO PLL	10,000000	23.809525	10.00 : 200.00
CAN 0				
CAN0 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN0 MIOCLK	External	0,100000		0.10 : 100.00
CAN 1				
CAN1 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN1 MIOCLK	External	0,100000		0.10 : 100.00
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100,000000	100.000000	0.10 : 250.00
FCLK_CLK1	IO PLL	200,000000	200.000000	0.10 : 250.00
FCLK_CLK2	IO PLL	200,000000	200.000000	0.10 : 250.00
FCLK_CLK3	IO PLL	50,000000	50.000000	0.10 : 250.00
System Debug Clocks				
TPIU	IO PLL	10,000000	0.010000	10.00 : 300.00
Timers				
WDT	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0				
TTC0 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1				
TTC1 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00

Figure 51: Clock Configuration

4.4. Generating the BitStream File and Exporting to SDK

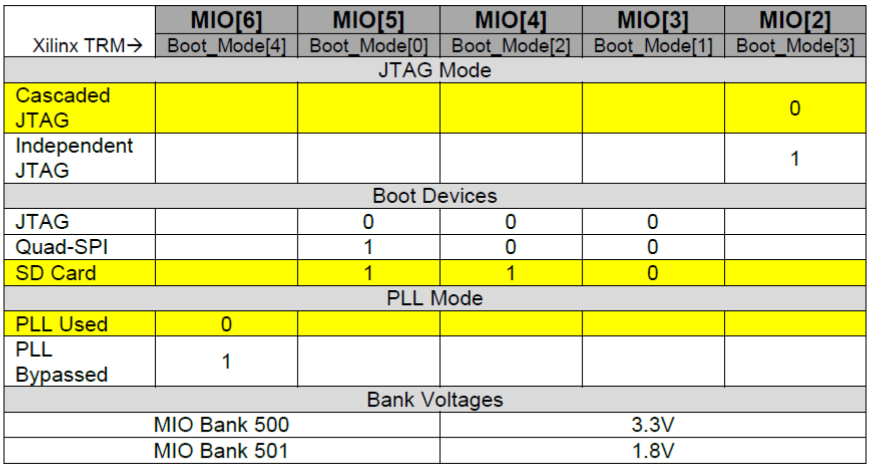
As in the previous chapter, after finishing the previous configuration, the **hardware platform** is completely **configured** and the project is prepared for the BitStream generation. Once again, the hardware platform has to be built and **exported to SDK** for being able to develop any application. Building includes the top level wrapper ge neration, the synthesis, the implementation and the BitStream generation.

The entire build process will create the “system.bit” file, but it will take approximately from 60 to 120 minutes depending on the PC’s capabilities, because of the **major complexity** of the hardware configuration file. Therefore, the hardware design is already finished; any **software** project associated with the hardware design can be created within SDK (e.g. the First Stage Boot Loader).

Remember that the **exported files** are the Hardware Platform Specification, “system.xml” (contains the hardware platform description for FSBL and BSP generation); the BitStream file, “system.bit”; “ps7_init.c”; “ps7_init.h”; “ps7_init.tcl”; and “ps7_init.html”.

4.5. Stage 0 or BootROM

As already occurred with the other Linux OS, it will be introduced on ZedBoard using the SD Card, hence the **MIO Configuration Modes** are **the same** as it was listed for the previous Linux OS:



	MIO[6]	MIO[5]	MIO[4]	MIO[3]	MIO[2]
Xilinx TRM→	Boot_Mode[4]	Boot_Mode[0]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

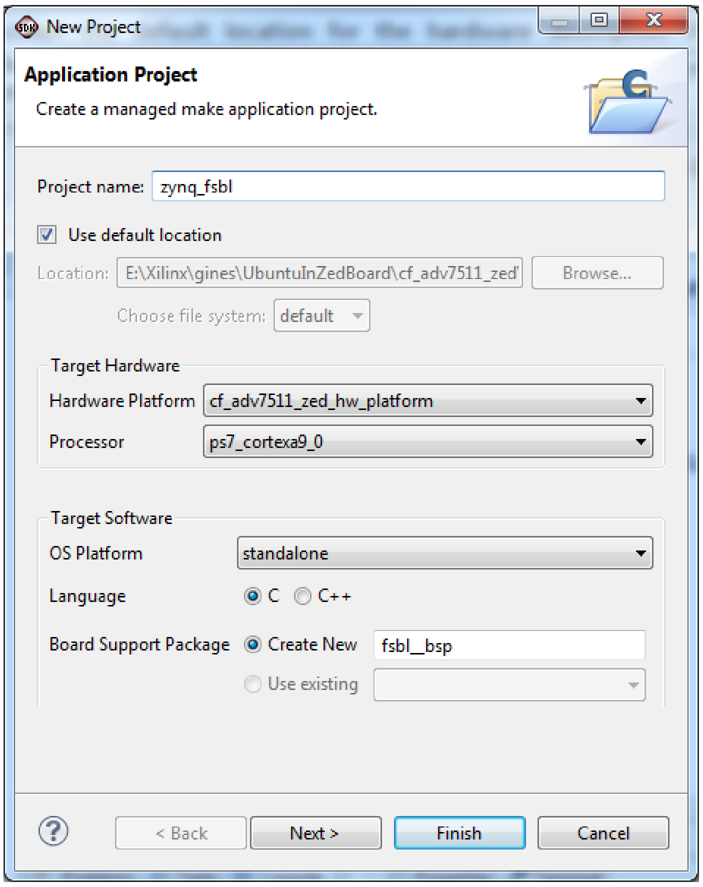
Figure 52: ZedBoard Configuration Modes

4.6. Boot Image, “boot.bin”

Once the BootROM is configured, the next step is creating the **boot file**. Once again, it will be only consisted on 3 files:

- The specific **First Stage Boot Loader** (FSBL) for ZedBoard.
- The specific **Programmable Logic Hardware BitStream** file for ZedBoard (optional).
- The U-Boot file, as **Second Stage Boot Loader**.

It is not necessary to explain again the **FSBL** and **BitStream** files, because the only difference with the previous Linux OS is that they are built from a different hardware configuration, but the build process and their purpose are the same.


New Project

Application Project

Create a managed make application project.

Project name: zynq_fsbl

☑ Use default location

Location: E:\Xilinx\gines\UbuntuInZedBoard\cf_adv7511_zed

Choose file system: default

Target Hardware

Hardware Platform: cf_adv7511_zed_hw_platform

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

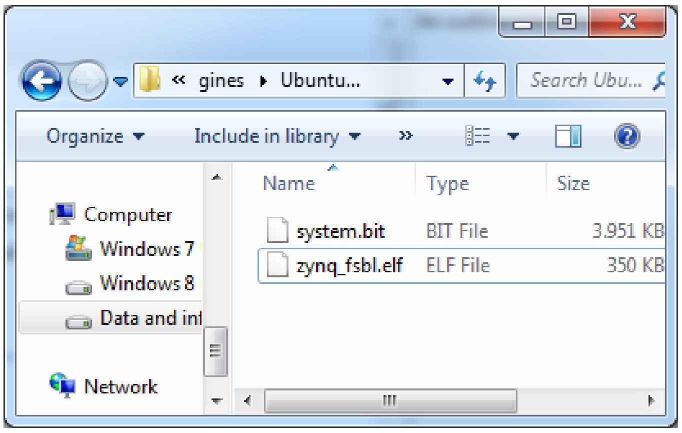
Language: ● C ○ C++

Board Support Package: ● Create New fsbl_bsp

○ Use existing

Figure 53: Setting the FSBL for Ubuntu

The FSBL and the BitStream files can be copied in a new folder, for example, called “UbuntuLinuxInZedboard”.



Name	Type	Size
system.bit	BIT File	3.951 KB
zynq_fsbl.elf	ELF File	350 KB

Figure 54: FSBL and BitStream Files Copied into UbuntuLinuxInZedBoard

On the other hand, the **U-Boot** file is the only file that should be **configured differently** to the previous chapter, because it must be configured in order to load the Root File System Image from the ext4 partition of the SD card, unlike the previous case. Nevertheless, the board is the same, ZedBoard.

The ZedBoard specific header references a second file where parameters common to all Zynq boards are defined. This header ("zynq_zed.h") and the **Zynq-parameter file** ("zynq-common.h") are found in …/u-boot-xlnx/include/configs/.

This second file must be opened in order to **edit the SD card boot ("sdboot") configuration line** which copies the root file system from the SD card to memory. By default, it is configured to load the "uramdisk.image.gz" file, like in the previous chapter, being defined to load the Linux Kernel image, the Device Tree Binary file and the compress Root File System image.

```
"sdboot=if mmcinfo; then " \
      "run uenvboot; " \
      "echo Copying Linux from SD to RAM... && " \
      "fatload mmc 0 0x3000000 ${kernel_image} && " \
      "fatload mmc 0 0x2A00000 ${devicetree_image} && " \
      "fatload mmc 0 0x2000000 ${ramdisk_image} && " \
      "bootm 0x3000000 0x2000000 0x2A00000; " \
```

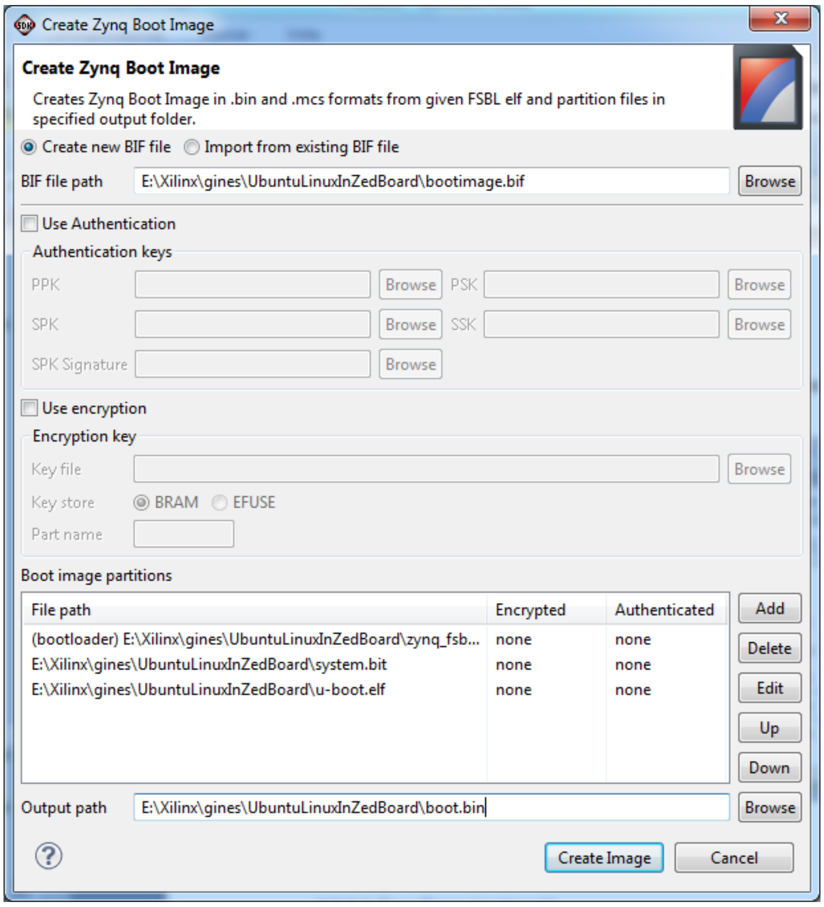
Command Window 4: Default SD Card Boot Configuration on Embedded Linux OS

Nevertheless, loading the Root File System from the ext4 partition is required. Thus, the previous file is modified and the result is the following:

```
"sdboot=echo Copying Linux kernel from SD to RAM...RFS in ext4;" \
      "mmcinfo;" \
      "fatload mmc 0 0x3000000 ${kernel_image};" \
      "fatload mmc 0 0x2A00000 ${devicetree_image};" \
      "bootm 0x3000000 - 0x2A00000\0" \
```

Command Window 5: Default SD Card Boot Configuration on Linaro-Ubuntu

Finally, all the required files to build the boot file have been obtained. Therefore, the boot image can be created using the "**Bootgen**" tool as in the previous chapter and save in "UbuntuLinuxInZedboard". Remember the **importance** of the **images order**.


Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

[✓ Create new BIF file] Import from existing BIF file

BIF file path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\bootimage.bif

Use Authentication [ ]

Authentication keys: PPK, PSK, SPK, SSK, SPK Signature

Use encryption [ ]

Encryption key: Key file; Key store [✓ BRAM] EFUSE; Part name

Boot image partitions

File path	Encrypted	Authenticated
(bootloader) E:\Xilinx\gines\UbuntuLinuxInZedBoard\zynq_fsb...	none	none
E:\Xilinx\gines\UbuntuLinuxInZedBoard\system.bit	none	none
E:\Xilinx\gines\UbuntuLinuxInZedBoard\u-boot.elf	none	none

Output path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\boot.bin

Figure 55: Zynq Boot Image Generation Window in SDK

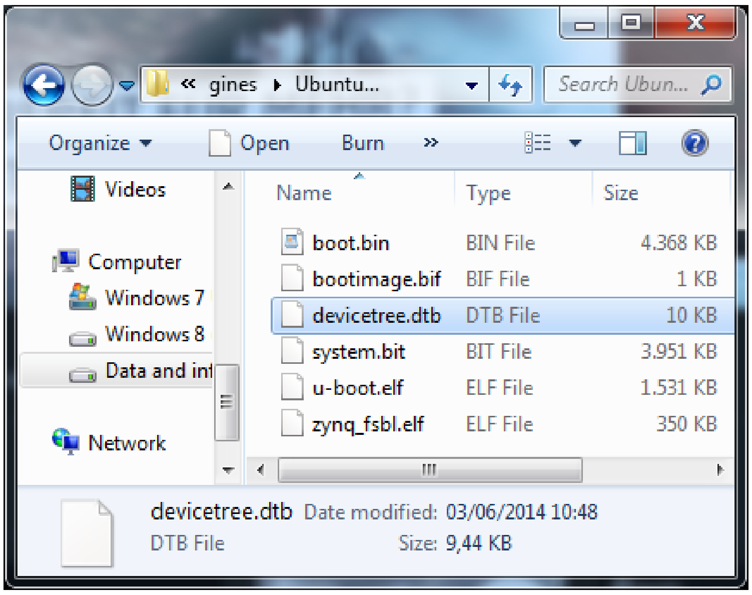
4.7. Device Tree Binary, “devicetree.dtb”

Remember that the Linux Kernel is a piece of embedded standalone software running on hardware, which has to know every detail about the hardware where it is running on. For this reason, it uses the **Device Tree Binary** file. Nevertheless, it is not necessary to explain again its building process, because the only difference with the previous **Linux OS** is that they are **built** from a different git repository Linux version, but the build process and their purpose are the **same**.

Linux-xlnx has been used to build the custom Linux OS. However, another Linux version will be used in this case. The new Linux version used is from **Analog Devices Inc**, because it incorporates the HDMI interface settings. Therefore, this Linux directory directly provides a DTS file with this configuration. It can be found in:

.../ubuntu/arch/arm/boot/dts/zynq-zed-adv7511.dts

Finally, the DTB file can be obtained by the “make ARCH=arm zynq-zed-adv7511.dtb” command and copied into “UbuntuLinuxInZedboard”.



Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
devicetree.dtb	DTB File	10 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
zynq_fsbl.elf	ELF File	350 KB

devicetree.dtb Date modified: 03/06/2014 10:48
DTB File Size: 9,44 KB

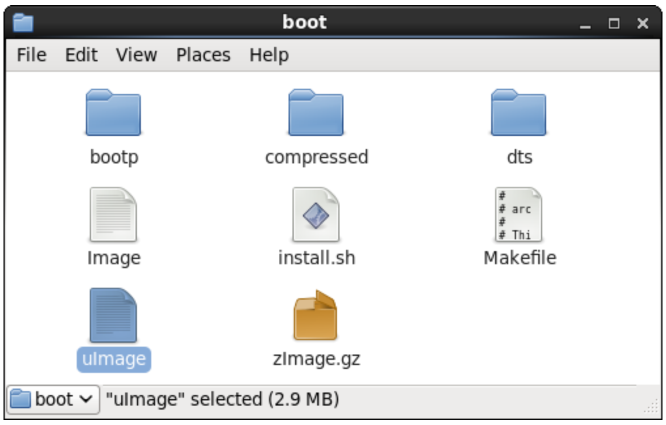
Figure 56: Device Tree Binary File in the "UbuntuLinuxInZedBoard" Folder

4.8. Linux Kernel File, "uImage"

Remember that the **kernel** is the lowest level of easily replaceable software that interfaces with the hardware in the device. It is responsible for interfacing all of the applications that are running in "user mode" down to the physical hardware, and allowing processes, known as servers, to get information from each other using inter-process communication (IPC).

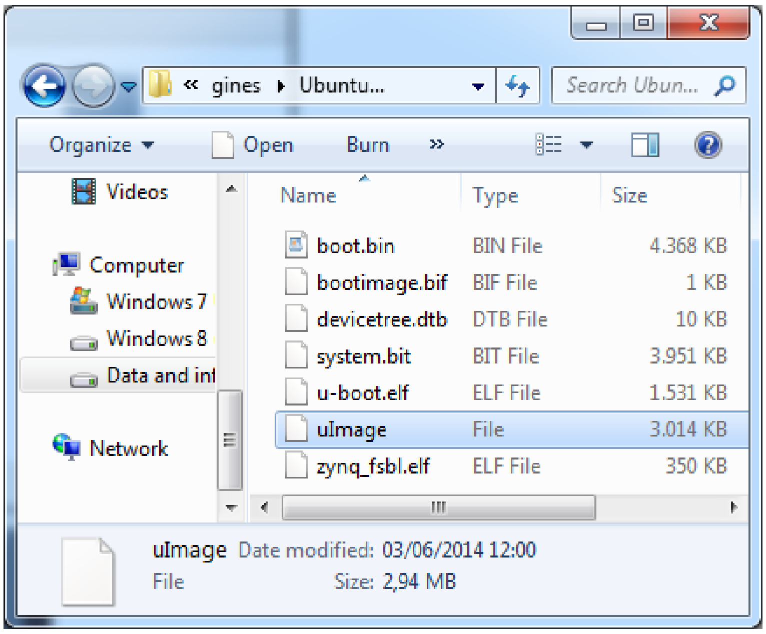
Despite the Linux version used is different, the build process is almost the same. The only difference is the configuration file. In the previous Linux OS was "xilinx_zynq_defconfig", while now is "**zync_xcomm_adv7511_defconfig**".

After configuring Linux, the "make ARCH=arm" command generates the "**zImage**" file. It has to be **converted into** a "**uImage**" file in order to allow to U-Boot to recognize it. This conversion can be carried out by using the "mkimage" tool, available in the U-Boot directory.



"uImage" selected (2.9 MB)

Figure 57: "uImage" Location Folder

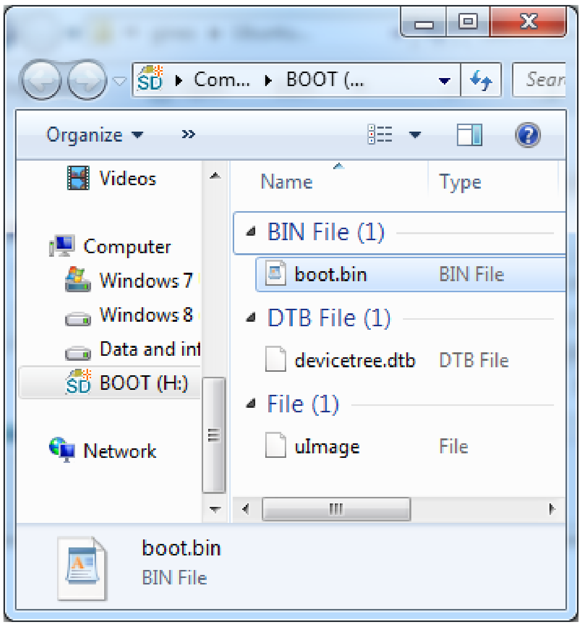


Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
devicetree.dtb	DTB File	10 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
uImage	File	3.014 KB
zynq_fsbl.elf	ELF File	350 KB

uImage Date modified: 03/06/2014 12:00
File Size: 2,94 MB

Figure 58: "uImage" in the "UbuntuLinuxInZedBoard" folder

Finally, the 3 required files for the FAT32 partition of the SD Card are ready.



Name	Type
BIN File (1)	
boot.bin	BIN File
DTB File (1)	
devicetree.dtb	DTB File
File (1)	
uImage	File

boot.bin
BIN File

Figure 59: "BOOT" Partition of the SD Card

4.9. Root File System Image

The FAT32 partition is ready, now it is only necessary to introduce the **Root File System** files into the ext4 partition. Remember that it is contains all operating system files.

This **section** is the one **which changes most** from the previous Linux OS, because it will be directly obtained from the Linaro-Ubuntu website and saved into the ext4 partition of the SD card. There are different Ubuntu versions available to download and directly implement into this partition. In this case, the selected Linaro-Linux version is the **Linaro 12.09 version**. It has to be downloaded, unzipped, and introduced into the ext4 partition.

Remember that the **ext4** partition is **not readable from Windows**, a Linux OS or Linux Virtual Machine has to be used to read and write in ext4 format.

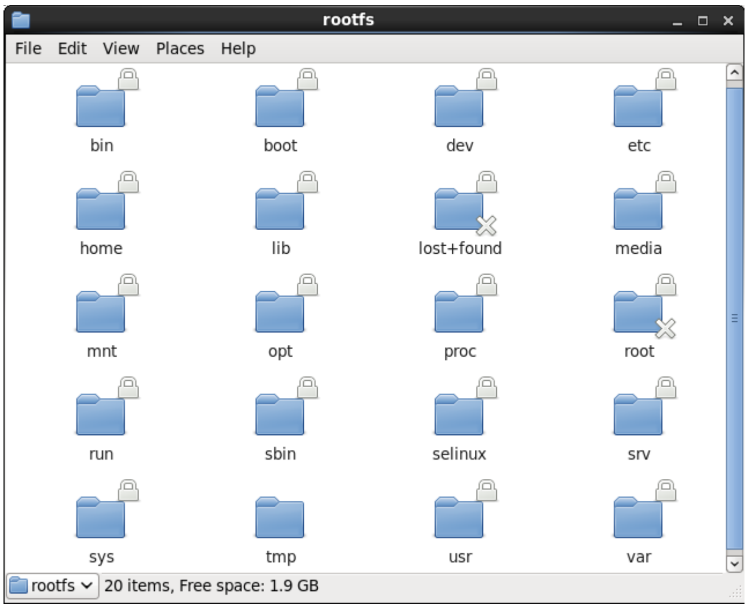


Figure 60: Linaro-Ubuntu 12.09 File System Files

4.10. Booting the Custom Embedded Linux OS

Once the Root File System Image is built, the Linaro-Ubuntu Linux OS can be booted on ZedBoard.

Connecting the **external peripherals** to the ZedBoard is **required**. At minimum, the keyboard, the mouse, the HDMI monitor, and the serial connection for the console are required. If some of **these peripherals** are **not connected**, the screen will show a error message and it will be **only possible** to use this OS thought the **serial connection** (i.e. using Tera Term).

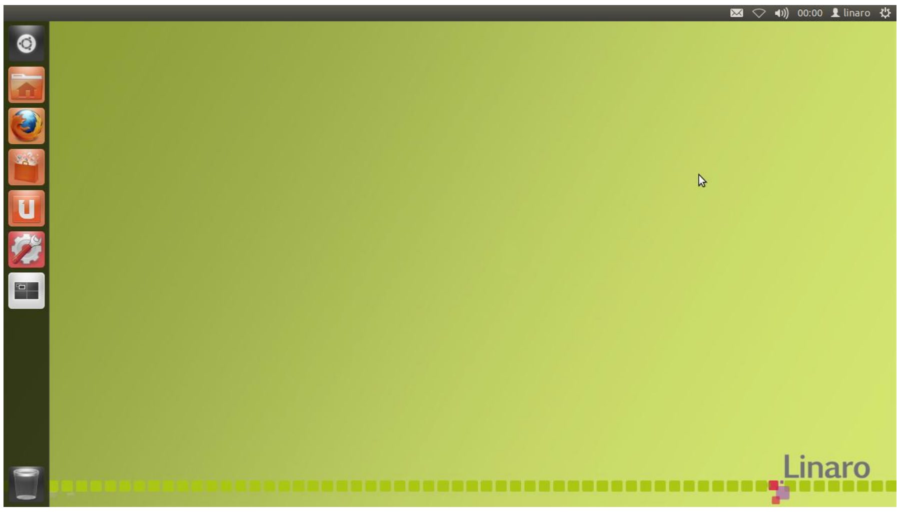


Figure 61: Linaro-Ubuntu 12.09 Appearance

Chapter 5: Summary and Conclusions

The results and conclusions of this work can be divided into these two parts:

- The development and implementation of a **custom Linux OS on ZedBoard**.
- The **differences** between implement a **not-graphical-user-interface custom Embedded Linux OS and** implement a complex HDMI **graphical user-interface GNU pre-designed Embedded Linux OS**.

5.1. Development and Implementation of a custom Linux OS on ZedBoard

Several actions, files and tools have been required to be able to develop and implement a specific OS into a specific board, in this case ZedBoard.

On one hand, these **required files** are responsible for all the tasks that ZedBoard needs to execute in a particular order to boot Linux OS on it. These tasks range from prepared and configure the PL and PS in the first stages to boot the operating system itself at last instance ultimately.

On the other hand, the entire set of **necessary actions** which have been necessary to configure and build these files throughout this thesis will be remembered and briefly summarized below. They will be also accompanied by the most relevant information and tools which should be known about them.

First, the **board** has been **selected**. ZedBoard has been the preferred board because of its performance and features. ZedBoard is not only a traditional FPGA, it is a complete SoC (System on a Chip), joining the PS (Processing System) with the PL (Programmable Logic).

Second, once the board is set, there were an endless number of different **Operating Systems** which could be **implemented** on it, depending on the needs of the end user. In this case, **two different** and opposing kinds of systems have been developed. On one hand, a "basic" not-graphical-user-interface custom Embedded Linux OS, designed exclusively for specific purposes. On the other hand, a complex HDMI graphical user-interface GNU pre-designed Embedded Linux OS, which can be used similarly to a laptop or PC.

Third, the **BootROM** or Stage 0 has been configured. It allows choosing among several boot modes; in this case, it has been configured to load the entire code from the SD Card. Remember that this stage is not-user configurable.

Next, the **ZedBoard required files** to implement the desired OS have been configured and built in order to be able to implement them into the board. These files have been configured specifically for the particular operating system and board. Nevertheless, their general purpose is the same, regardless of the OS or the board. These four-required files are showed below:

1. The **Boot Image** ("boot.bin"), which initializes the processor resources (FSBL file), configure the programmable logic (BitStream file), and loads the Linux kernel (U-Boot file).

2. The **Device Tree Binary File** ("devicetree.dtb"), which is obtained from the Device Tree Source File and loaded into the DDR memory by U-Boot. It allows the kernel to know every detail about the hardware in which it is working on.
3. The **Linux Kernel File** ("uImage"), which is also loaded into the DDR memory by U-Boot. It initializes the system hardware and mounts the root file system.
4. The **Root File System Image** ("uramdisk.image.gz" or another partition with the OS files and folders), which can be a small custom image which is loaded into the DDR memory or can be a complete pre-configured OS which contains all necessary files and libraries to work as a normal PC.

The two different Linux OS offer GNU software in which all the **advantages and disadvantages** of implementing it on a SoC board, such as ZedBoard, have progressively emerged throughout the entire thesis. Fortunately, the number of advantages has been far outweighs than the number of disadvantages. They will be listed as a whole, to demonstrate the convenience of knowing how implement a Linux OS on ZedBoard:

+ Today, **great variety of devices** (Smartphones, Smart TVs, etc) have small integrated embedded systems; being an attractive field for developers.
+ **ZedBoard** is not only a FPGA, but also it is a complete SoC IC. Therefore, it has different memories types, a more powerful processing system, a great variety of peripherals, etc. In addition, ZedBoard provides an **ARM** processor, which is the most common processor in the devices mentioned above.
+ Increasingly, these devices are being programmed with a **Linux OS**, because of the Linux advantages showed in the first chapter, highlighting their no cost.
+ Thanks to the facilities which are provided by **BuildRoot**, the process is greatly simplified, allowing loading default configuration files, changing or adding only the files or libraries which are going to be customized respect to the standard configuration. If BuildRoot is not used, the required time to carry out the whole process will be increased exponentially.
+ There are a **countless** number of GNU **directories** which have **different** but interesting **configurations** (e.g. pre-configured for HDMI connection). It can be necessary different configuration process for each one, but they provide greatest option diversity. Nevertheless, the configuration usually has no relevant differences between them.
+ There are a **great variety of tutorials, information and** community **support**, helping to know how configure and develop the desired files.

– Nevertheless, if there are a lot of information, there are also **a wide range of variants and concepts to learn and understand**, requiring a **large amount of time** to fully understand all of them before starting to develop a specific OS; beginning with the hardware architecture study and the development flow for the Zynq-7000 AP SoC comprehension; fend for oneself with the Xilinx development tools; understanding the Linux configuration and build flow as well as U-Boot configuration and building process (differences depending on the board, the OS, if the files are loaded from the SD Card or the flash memory, choosing between a "zImage" or "uImage" Linux kemel file, etc).

- There are **several ways** to perform the **same task**, which can be confusing at the beginning of the project.
- There are several tutorials and guides to perform and build the required files, about "how build them", but is **not clearly explained** why these tasks are performed, "what and why must be done", requiring also a large amount of time to really understand this.
- As with any technology, the **code** is **outdated** in a small time period; therefore it must be periodically updated, as well as the tools required for it.

Despite the drawbacks, is clear that the amount of benefits is clearly higher.

Finally, Linux is ready to be used on ZedBoard; a host PC can interchange information with ZedBoard by using a serial port connection program, such as Tera Term. Nevertheless, in the Ubuntu case, this program is not necessary if a keyboard, a mouse and a HDMI monitor are connected to the board. Therefore, the **ZedBoard** can be **switched on**.

5.2. Differences between a Custom Linux OS and an Ubuntu Linux OS

In this project, two variants of Linux OS have been developed, analyzing the advantages and disadvantages that each one has. Which one must be used depends on the end user who will use it and depends on its purpose.

On one hand, if a very specific application is needed, if there are a limited amount of resources, and/or if a very fast response of the board is needed (e.g. in RTOS), a custom Linux **OS** is the best option. In this case, a **not-graphical**-user-interface **custom Embedded Linux OS** has been developed, which can be designed exclusively for the specific, desired purpose. It can be configured in order to use only the minimum required resources, not wasting memory and resources in peripherals which are not going to be used.

Therefore, the **advantages and disadvantages** of this king of systems are listed below:

+ **Fewer resources** needed, just the necessary and required.
+ The system required **memory** is **less**.
+ **Less** physical **size** of the device, because of the least required resources.
+ **Greater speed**, because it requires to manage fewer resources and less programs at the same time.
+ **Flash memory** possibility, due to its less required memory.
+ **RTOS** possibility, the system does not need a complete user-interface, requiring fewer resources and becoming in a faster system.

- If the system **target changes**, the whole **system** must be complete re-configured and **re-build**.
- It is **not** possible to use a **pre-configured** generic system, it can be used a default configuration, but it is necessary configure it.

- It **cannot** be used as a **generic system**; it can only perform its specific functions.
- It usually has **not** an **easy user-interface**, getting harder to use.
- It requires more time to **configure** the whole system.
- More difficult to **configure** it (i.e. configuring the whole system is required).

On the other hand, if a more generic used is desired, if there are an enough amount of resources, if a wide range of applications is required and/or if it is not necessary a RTOS, a complex graphical user-interface pre-designed Embedded Linux OS is the best option. In this case, a complete-HDMI-user-interface generic **Ubuntu Desktop Linux OS** has been implemented, which can be configured to a wider range of applications.

Therefore, the **advantages and disadvantages** of this king of systems are listed below:

+ If the system **targets change**, it is **not** necessary to **change** the **whole system**, it is possible to only add the new requirements. It is not necessary to remove peripherals or delete programs unless it required larger memory or resources.
+ It is possible to use a **pre-configured** generic system, configuring and adding only the functions which the default system does not have.
+ It **can** be used as a **generic system**, like a generic laptop can do.
+ It usually has an **easy user-interface**, getting easy to use for everyone.
+ Faster to **configure** it, if a pre-configured OS is used.

- **More resources** are needed, because it is necessary to control a greater number of devices and execute a greater number of applications.
- The system required **memory** is **higher**.
- **Greater** physical **size** of the device, because of the higher required resources.
- **Lower speed**, because it requires managing a wide range of applications and resources at the same time.
- **Small possibility of flash memory** use, due to its higher required memory.
- **No-RTOS** possibility, due to its wide range of required resources and applications, becoming in a slower system.

As a conclusion, both of them have **great advantages**, which make them useful for **different types of applications**. Nevertheless, they also have some drawbacks which must be considered depending on the desired application which is going to be used for. Therefore, it is necessary to know the specifications which the end-user desires.

Chapter 6: Bibliography

Using the entire thesis:

[1] http://www.oa.upm.es/21488/1/PFC_ALVARO_BUSTOS_BENAYAS.pdf
[2] http://www.wiki.xilinx.com/
[3] http://www.wikipedia.org/
[4] http://www.xilinx.com/
[5] http://www.zedboard.org/
[6] http://www.zedboard.org/course/introduction-zynq
[7] http://www.zedboard.org/documentation/1521
[8] http://www.zedboard.org/product/zedboard

Chapter 1:

[9] http://www.free-electrons.com/doc/training/embedded-linux/slides.pdf
[10] https://www.ibm.com/developerworks/linux/library/l-embl/

Chapter 2:

[11] http://en.wikibooks.org/wiki/Embedded_Systems
[12] http://git.denx.de/?p=u-boot.git;a=blob_plain;f=README;hb=HEAD
[13] http://www.abdulet.net/?p=530
[14] http://www.buildroot.uclibc.org/
[15] http://www.elinux.org/Zedboard
[16] http://www.embeddedsoftwarestore.com/
[17] http://www.github.com/Xilinx
[18] http://www.stlinux.com/u-boot
[19] http://www.synnick.blogspot.co.at/2012/02/sistemas-embebidos-y-ejemplos.html
[20] http://www.webopedia.com/TERM/E/embedded_system.html
[21] http://www-pnp.physics.ox.ac.uk/

Chapter 3:

[22] http://www.csee.umbc.edu/
[23] http://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/
[24] http://www.howtosetproxiesinyourbrowser.blogspot.co.at/2010/12/what-is-difference-between-zimage-bz.html
[25] http://www.stackoverflow.com/
[26] http://www.zedboard.org/content/zedboard-create-planahead-project-embedded-processor

Chapter 4:

[27] http://git-scm.com/docs/git-checkout
[28] http://wiki.analog.com/
[29] http://www.linaro.org/
[30] http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/
[31] http://www.releases.linaro.org/

Appendix 1: Guide – Linux on ZedBoard step by step

1. Overview

First, note that this Appendix is closely related to "Chapter 3: Custom Embedded Linux OS on ZedBoard" and "Chapter 4: Ubuntu Linux OS on ZedBoard". These chapters are complementary to this appendix and vice versa; they will deal with the development and implementation of the Linux OS. Nevertheless, they will do it in a different form.

These chapters are focus on the idea, on the concepts, on answering the questions "what must be do", "why must be do" and "which ways are there available" to develop and implement the Linux OS. Nevertheless, this appendix is focused on the exactly steps to achieve the target, on answering "how must be exactly done to develop and implement it".

Therefore, it is strongly recommended to read and understood these chapters before performing this appendix.

As in these chapters, the final achievement of this guide is to implement a custom Linux Operating System (OS) integrated with a specific PL peripheral. How to create, configure, build and implement an Embedded Linux OS on ZedBoard will be showed in detail in this appendix.

Remember that **Xilinx** and its whole family of programs will be the main tool for achieving this goal. Nevertheless, these tools are not enough, some **GNU tools** are also required to get this achievement. Therefore, before continuing this appended, "Appendix 2: Prerequisites" must be carried out, where all the required programs will be downloaded and pre-configured.

In this guide will be explained in detailed two different options to build the Linux OS:

- Developing a **custom Embedded Linux OS** which will be designed exclusively for the specific purposes (for the specific PL peripheral).
- Implementing a **GNU pre-designed Embedded Linux OS** in which the specific PL peripheral can be easily added.

They have different but interesting and important advantages, as it was showed in the third and fourth chapters. Because of this, both of them will be developed and implemented on ZedBoard before selecting which one will be finally chosen to implement the PL peripheral.

Finally, before performing this appendix, it will be shown which programs are required and how install them in "Appendix 2: Prerequisites".

2. Custom Embedded Linux OS on ZedBoard

2.1. Overview

As already mentioned in the third chapter, the aim is to build a custom Embedded Linux OS on ZedBoard. For this purpose, four files are required in the SD Card to start this Linux OS on ZedBoard:

1. The **Boot Image** (“boot.bin”), which consists on the following files.
    a. The **FSBL** (First Stage Boot Loader).
    b. The **Programmable Logic Hardware BitStream** (optional).
    c. The **U-Boot** (Second Stage Boot Loader).
2. The **Device Tree Binary File** (“devicetree.dtb”).
3. The **Linux Kernel File** (“uImage”).
4. The **Root File System Image** (“uramdisk.image.gz”).

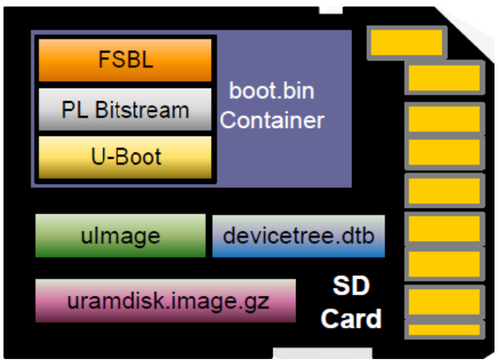


Figure 62: Files Required in the SD Card

Therefore, the programs and steps to create, configure, build and implement a “Basic” Embedded Linux Operating System with BuildRoot on ZedBoard are summarized below.

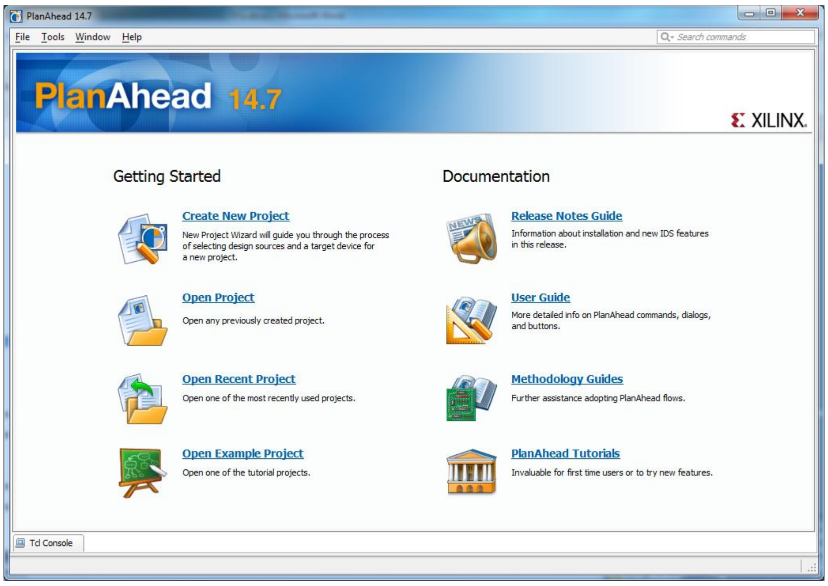
1. **PlanAhead:**
    a. Creating a PlanAhead Project.
    b. Adding an embedded source.
2. **Xilinx Platform Studio (XPS):**
    a. Configuring the Hardware Platform (e.g. peripherals, clocks, DDR3 memory, etc).
3. **PlanAhead:**
    a. Design Constraints.
    b. Top HDL Module.
    c. Hardware Platform Building.
4. **Software Development Kit (SDK):**
    a. Creating the FSBL (First Stage Boot Loader).
5. **BuildRoot:**
    a. Configuring the OS.
    b. Building the custom OS.
    c. Obtaining the U-Boot file, the Device Tree Binary file, the Kernel Image and the Root File System Image.

6. **Software Development Kit (SDK):**
   a. Creating the boot image ("boot.bin") from the FSBL, the BitStream file and the U-Boot file.
7. **SD Card:**
   a. Copying the four files in the SD Card.
8. **Tera Term:**
   a. Running Linux on ZedBoard.

2.2. Creating a New Project and Adding Embedded Sources

2.2.1. PlanAhead

Opening PlanAhead and clicking "Create New Project":



New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open any previously created project.

Open one of the most recently used projects.

Open one of the tutorial projects.

Information about installation and new IDS features in this release.

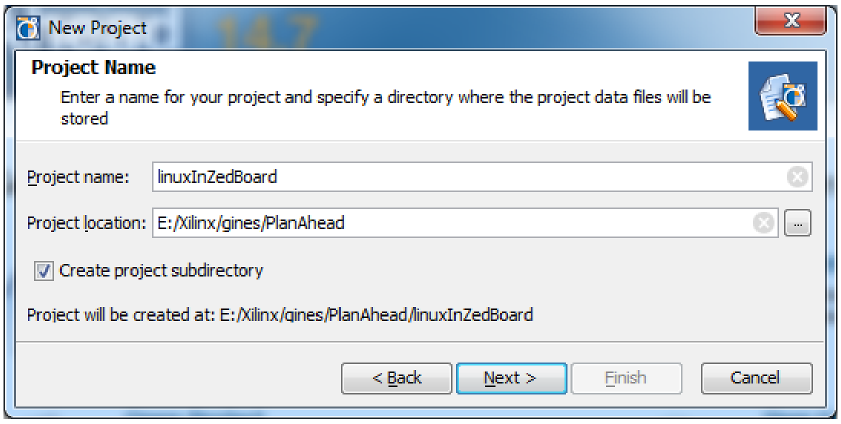
More detailed info on PlanAhead commands, dialogs, and buttons.

Further assistance adopting PlanAhead flows.

Invaluable for first time users or to try new features.

Figure 63: PlanAhead 14.7 Program

Click "Next" and set the Project Name and Project Location. This action will create a PlanAhead Project file (*.ppr) and locate it in a subdirectory with the same project name within the specified project location:



Project Name

Enter a name for your project and specify a directory where the project data files will be stored

Project name: linuxInZedBoard

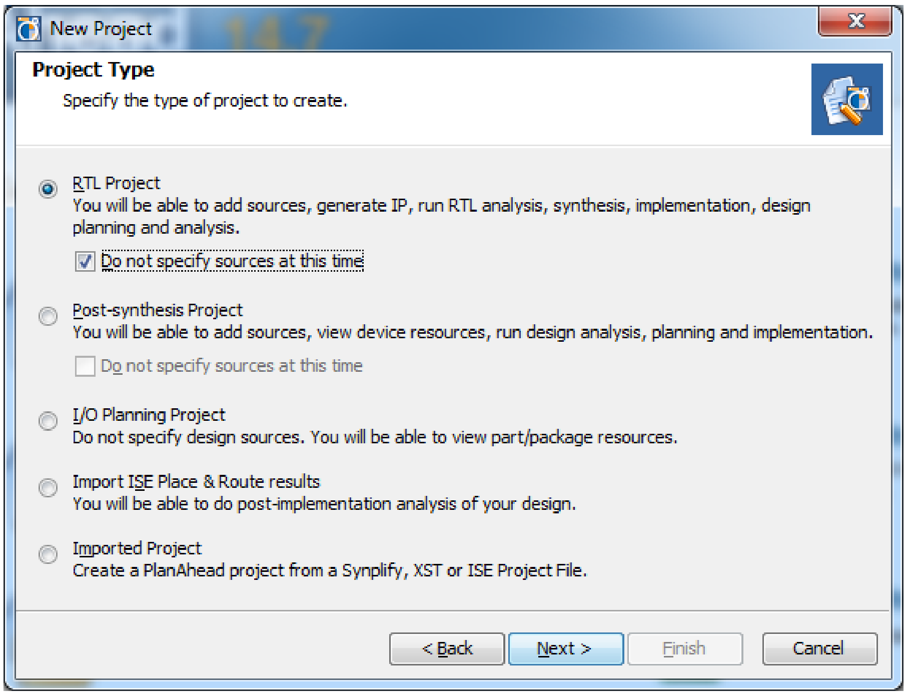
Project location: E:/Xilinx/gines/PlanAhead

[✓] Create project subdirectory

Project will be created at: E:/Xilinx/gines/PlanAhead/linuxInZedBoard

Figure 64: Project Name Window

Several types of projects can be created. A Register Transfer Level (RTL) project is required to manage the entire System on Chip design flow. Clicking "Do not specify sources at this time" because the sources will be imported later:


New Project

Project Type

Specify the type of project to create.

(●) RTL Project
You will be able to add sources, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
[✓] Do not specify sources at this time

( ) Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.
[ ] Do not specify sources at this time

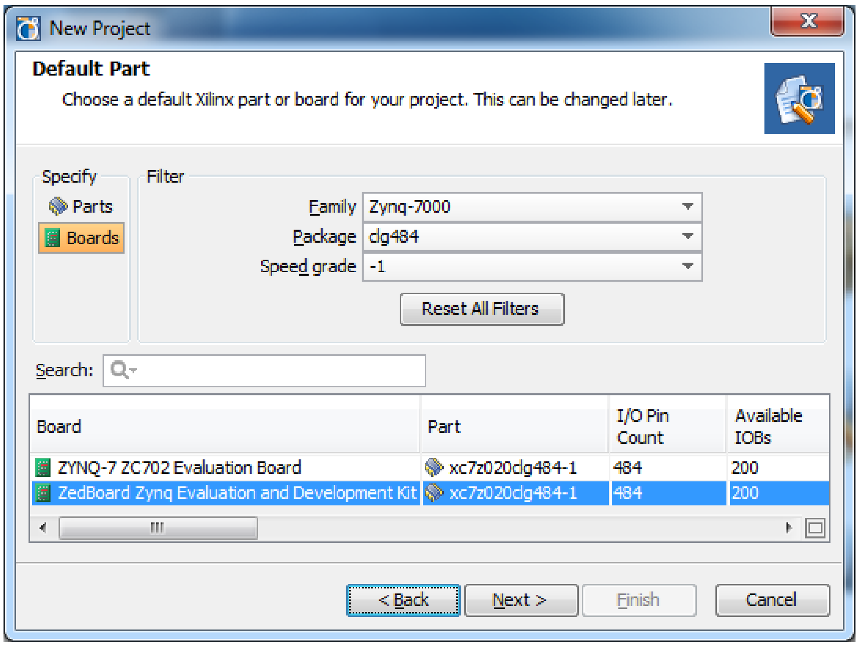
( ) I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

( ) Import ISE Place & Route results
You will be able to do post-implementation analysis of your design.

( ) Imported Project
Create a PlanAhead project from a Synplify, XST or ISE Project File.

Figure 65: Project Type Window

The next step is selecting the target device. This can be done by specifying a specific part or by selecting a development board. In "Boards", select Family Zynq-7000, Package clg484 and Speed grade -1. In "Parts", this is equivalent to selecting Zynq-7000 on Family and Sub-Family sections, Package clg484, Speed grade -1 and Temp grade C.


New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

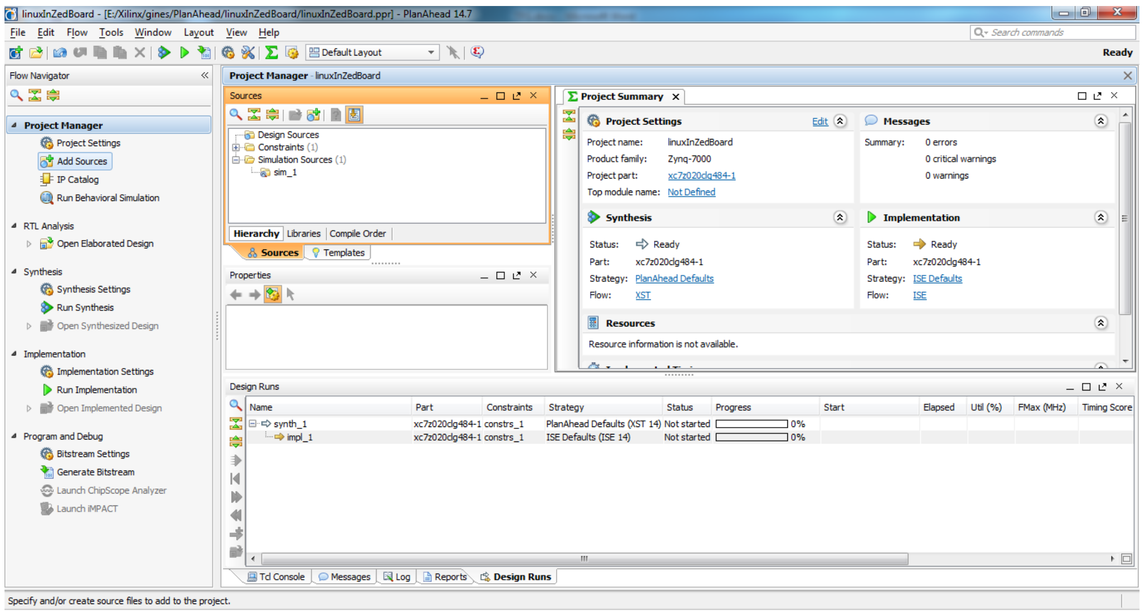
Specify: Parts, Boards (selected)

Filter
Family: Zynq-7000
Package: clg484
Speed grade: -1

Board	Part	I/O Pin Count	Available IOBs
ZYNQ-7 ZC702 Evaluation Board	xc7z020clg484-1	484	200
ZedBoard Zynq Evaluation and Development Kit	xc7z020clg484-1	484	200

Figure 66: Board/Part Selection

A project summary is displayed and "Finish". The current project is blank. Clicking in "Add Sources":



Project Settings

Project name:	linuxInZedBoard
Product family:	Zynq-7000
Project part:	xc7z020clg484-1
Top module name:	Not Defined

Messages

Summary:	0 errors
	0 critical warnings
	0 warnings

Synthesis

Status:	Ready
Part:	xc7z020clg484-1
Strategy:	PlanAhead Defaults
Flow:	XST

Implementation

Status:	Ready
Part:	xc7z020clg484-1
Strategy:	ISE Defaults
Flow:	ISE

Resources

Resource information is not available.

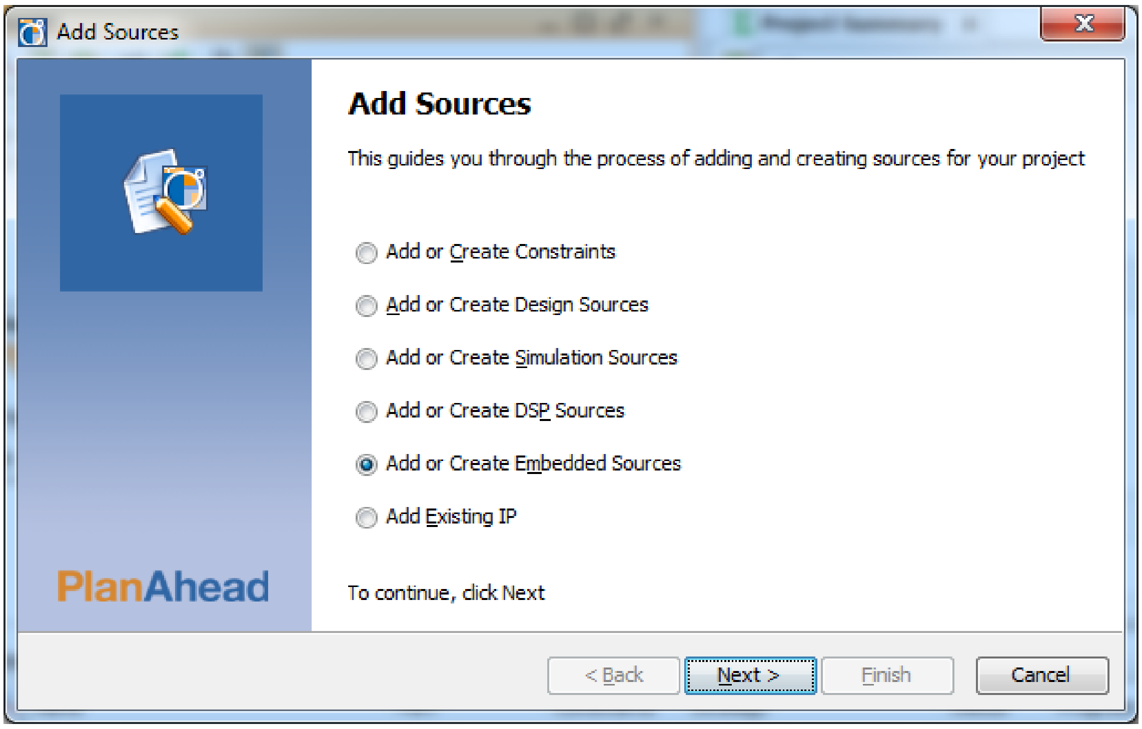
Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timing Score
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	Not started	0%					
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	Not started	0%					

Specify and/or create source files to add to the project.

Figure 67: PlanAhead 14.7 Program

To access the ARM processing system, an Embedded Source has to be added. Once the Embedded Source is created configuring the embedded system will be allowed:



Add Sources

This guides you through the process of adding and creating sources for your project

( ) Add or Create Constraints
( ) Add or Create Design Sources
( ) Add or Create Simulation Sources
( ) Add or Create DSP Sources
[✓ Add or Create Embedded Sources]
( ) Add Existing IP

To continue, click Next

Figure 68: Add Sources Window ½

An embedded source is created in "Create Sub-Design..." and is called "system" (generic name usually used):

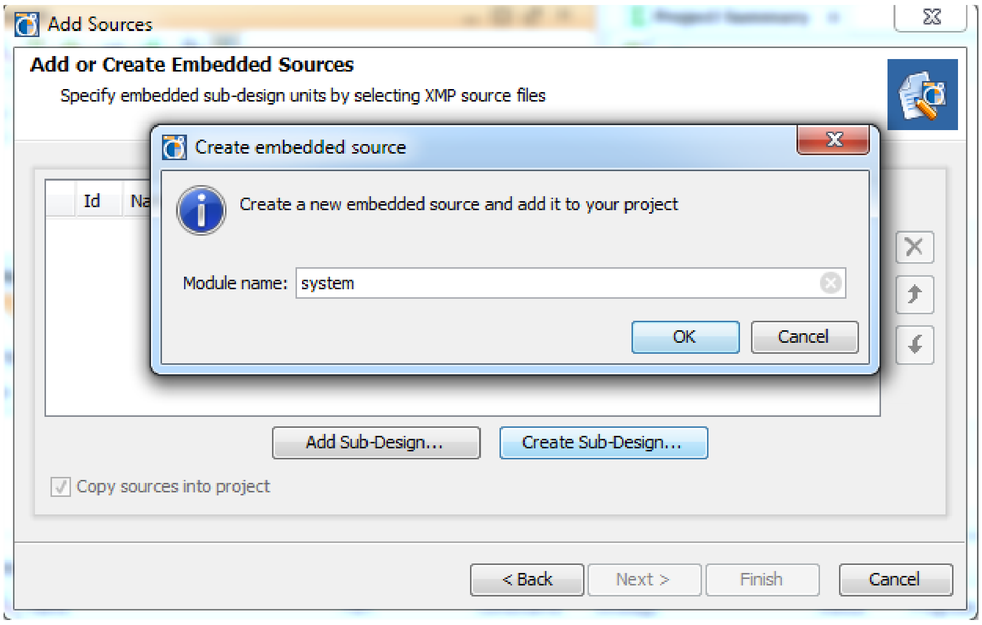
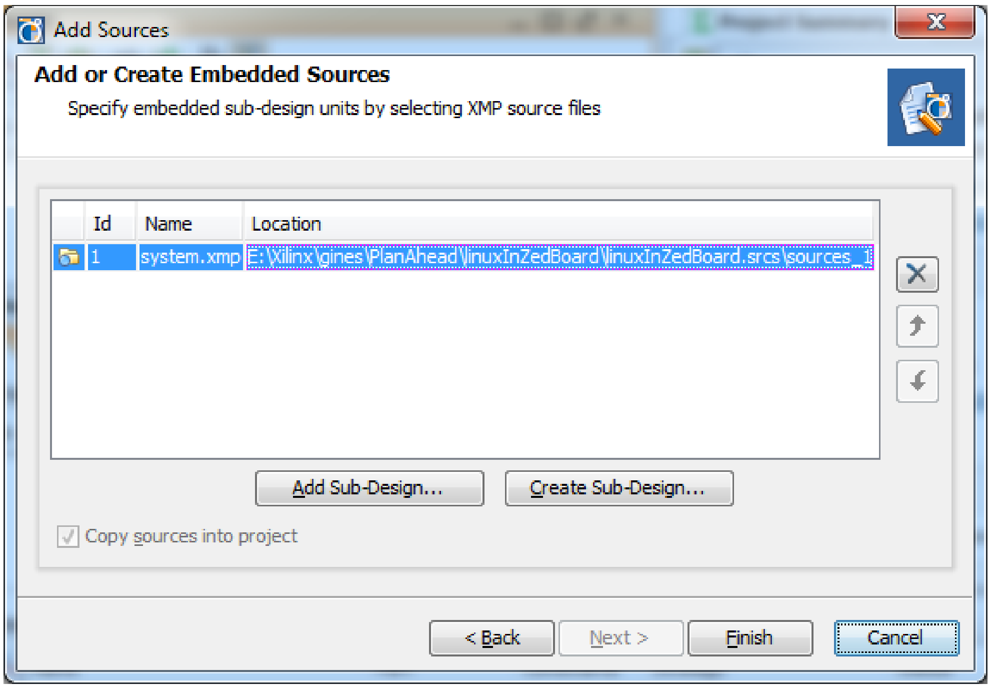

Add Sources
Add or Create Embedded Sources
Specify embedded sub-design units by selecting XMP source files
Create embedded source
Create a new embedded source and add it to your project
Module name: system
OK
Cancel
Id
Add Sub-Design...
Create Sub-Design...
Copy sources into project
< Back
Next >
Finish
Cancel

Figure 69: Create Embedded Source Window

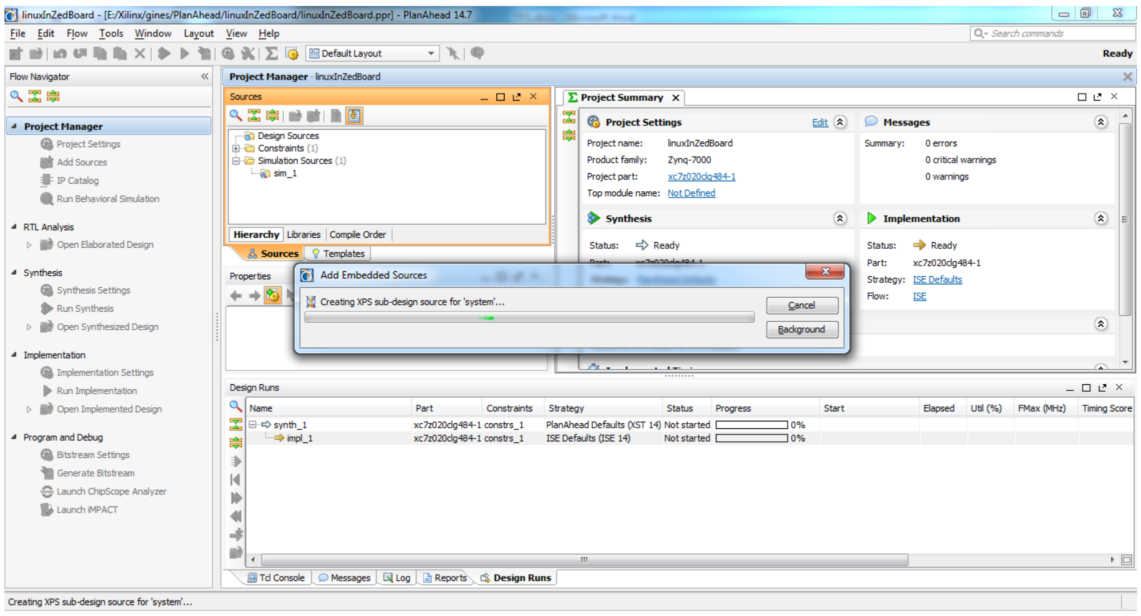
A Xilinx Microprocessor Project (XMP) file will be created. This file (system.xmp) is the top-level file descriptor of the embedded system. All project information is stored in the XMP file which is read by XPS and it is graphically showed. Therefore, system.xmp will be the embedded source file. Clicking Finish, PlanAhead will integrate the module in the sources of the project and create the required files for the project:


Add Sources
Add or Create Embedded Sources
Specify embedded sub-design units by selecting XMP source files
Add Sub-Design...
Create Sub-Design...
Copy sources into project
< Back
Next >
Finish
Cancel

Id	Name	Location
1	system.xmp	E:\Xilinx\gines\PlanAhead\linuxInZedBoard\linuxInZedBoard.srcs\sources_1

Figure 70: Embedded Source Added

XPS is automatically launched:


linuxInZedBoard - [E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.ppr] - PlanAhead 14.7

Project Settings

Project name:	linuxInZedBoard
Product family:	Zynq-7000
Project part:	xc7z020clg484-1
Top module name:	Not Defined

Messages

Summary:	0 errors
	0 critical warnings
	0 warnings

Implementation

Status:	Ready
Part:	xc7z020clg484-1
Strategy:	ISE Defaults
Flow:	ISE

Add Embedded Sources

Creating XPS sub-design source for 'system'...

Design Runs

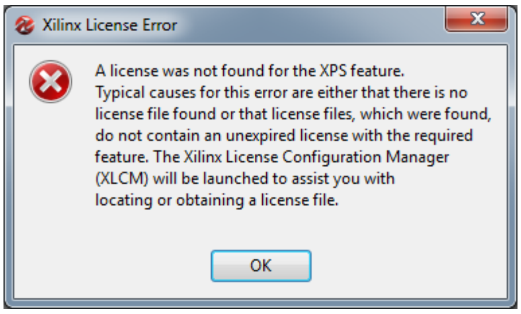
Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timing Score
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	Not started	0%					
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	Not started	0%					

Creating XPS sub-design source for 'system'...

Figure 71: PlanAhead Launch XPS

2.2.2. Xilinx Platform Studio (XPS)

After XPS is opened, an error is seen with the WebPACK license in Xilinx 14.7, this is a bug. WebPACK includes the Zynq XPS license; simply click “OK” and Close the Xilinx License Configuration Manager after it opens:

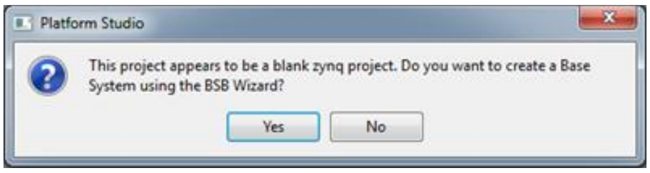

Xilinx License Error

A license was not found for the XPS feature.
Typical causes for this error are either that there is no license file found or that license files, which were found, do not contain an unexpired license with the required feature. The Xilinx License Configuration Manager (XLCM) will be launched to assist you with locating or obtaining a license file.

OK

Figure 72: Xilinx License Bug

The embedded source “system” is now open in XPS. A dialog box will open asking if the developer wants a Base System using BSB (Base System Builder) wizard to be created. Although this is a very handy tool that can save a lot of steps in other applications, it won”t be used here. Click “No”:

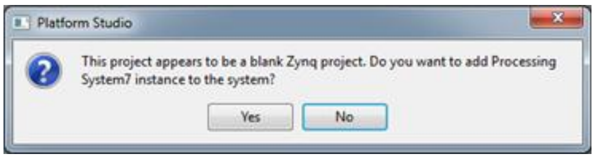

Platform Studio

This project appears to be a blank zynq project. Do you want to create a Base System using the BSB Wizard?

Yes No

Figure 73: Wizard Request in XPS

XPS ask for adding Processing System7 to the system. Click “Yes”:

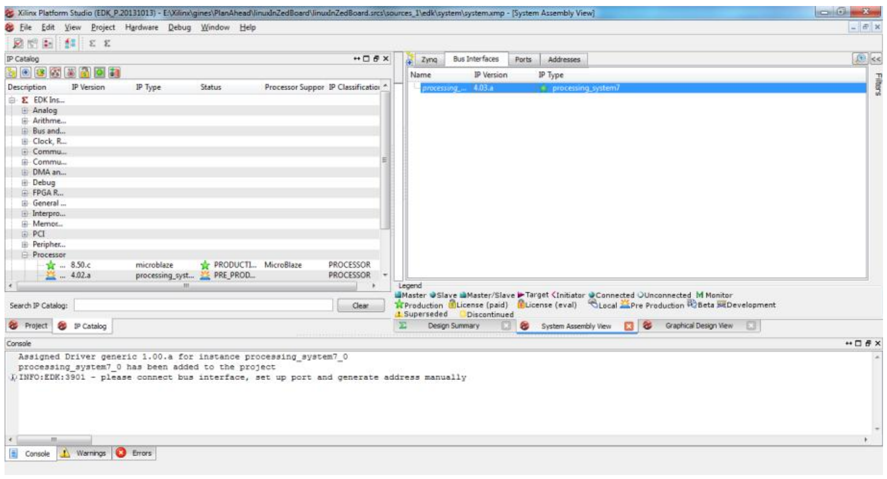


This project appears to be a blank Zynq project. Do you want to add Processing System7 instance to the system?

Figure 74: PS7-Adding Request in XPS

As already know, the Processing System 7 IP is the software interface around the Zynq Processing System. As known, Zynq-7000 family consists on a system-on-chip integrated processing system (PS) and a Programmable Logic (PL) unit. The Processing System 7 IP acts as a logic connection between the PS and the PL while assisting users to integrate custom embedded IPs with the processing system using Xilinx Platform Studio (XPS).

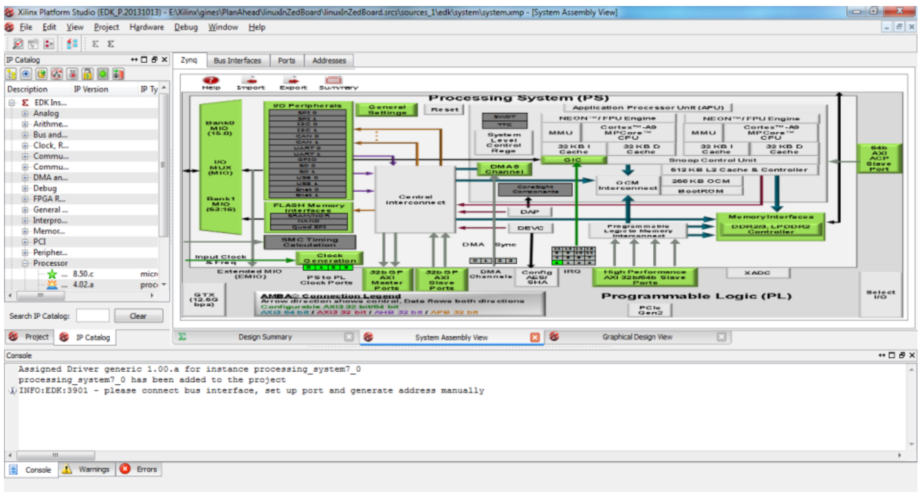
Some releases of Xilinx don”t offer this choice. The Processing System can be added by right clicking and “Add IP” in it in the processor section. Now, the processor system can be notice that was added in the “Bus Interfaces” tab:



```
Assigned Driver generic 1.00.a for instance processing_system7_0
processing_system7_0 has been added to the project
INFO:EDK:3901 - please connect bus interface, set up port and generate address manually
```

Figure 75: PlanAhead 14.7 Program

With the lower XPS System Assembly View tab selected, click on the upper Zynq tab:



```
Assigned Driver generic 1.00.a for instance processing_system7_0
processing_system7_0 has been added to the project
INFO:EDK:3901 - please connect bus interface, set up port and generate address manually
```

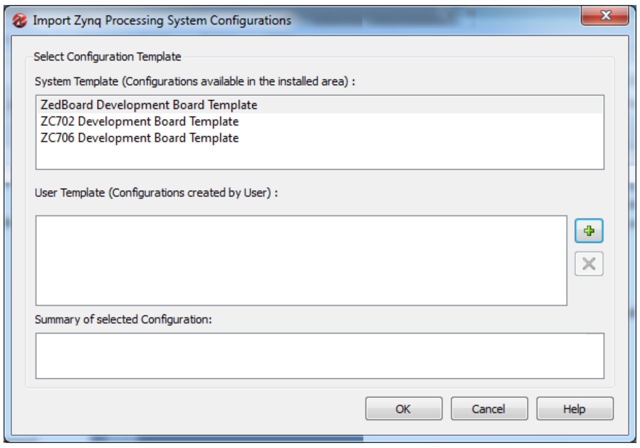
Figure 76: XPS after Adding the PS7

This PS is completely unconfigured, as indicated by all the I/O Peripherals being gray (none selected). All PS features are in their default state, ready to be customized, which is what it is going to be made. The green blocks are the configurable items by clicking in them.

The configuration of the peripherals could be made one by one or could be imported from a configuration file. ZedBoard.org provided a default configuration for the ZedBoard called "PS7 Configuration Definition (XML)", which can be downloaded here:

http://zedboard.org/documentation/1521

Once it is downloaded, clicking in "Import":



Import Zynq Processing System Configurations

Select Configuration Template

System Template (Configurations available in the installed area) :

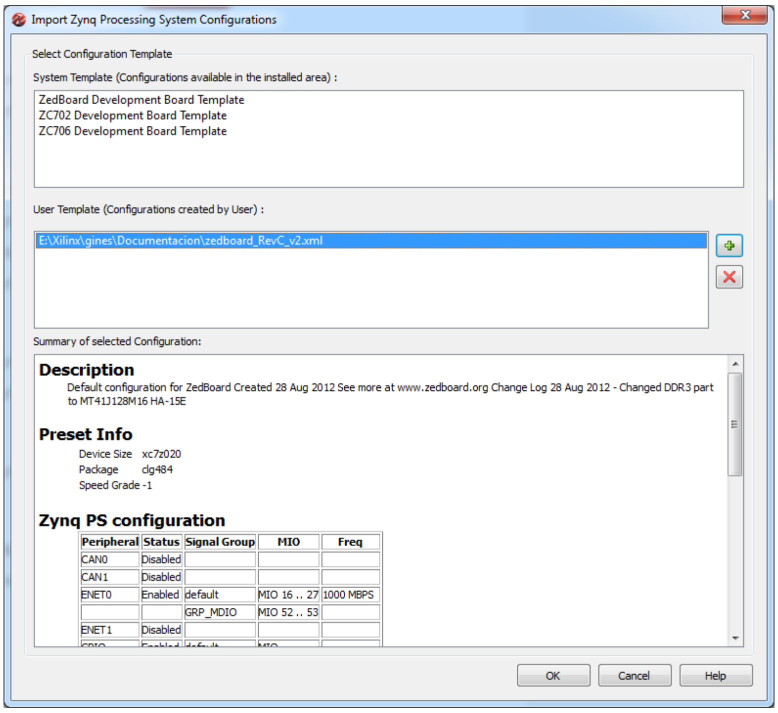
ZedBoard Development Board Template
ZC702 Development Board Template
ZC706 Development Board Template

User Template (Configurations created by User) :

Summary of selected Configuration:

Figure 77: Import PS Configuration Window 1/2

Choosing "ZedBoard Development Board Template", clicking in the green symbol and adding the required configuration file:



Import Zynq Processing System Configurations

Select Configuration Template

System Template (Configurations available in the installed area) :

ZedBoard Development Board Template
ZC702 Development Board Template
ZC706 Development Board Template

User Template (Configurations created by User) :

E:\Xilinx\gines\Documentacion\zedboard_RevC_v2.xml

Summary of selected Configuration:

Description

Default configuration for ZedBoard Created 28 Aug 2012 See more at www.zedboard.org Change Log 28 Aug 2012 - Changed DDR3 part to MT41J128M16 HA-15E

Preset Info

Device Size	xc7z020
Package	clg484
Speed Grade	-1

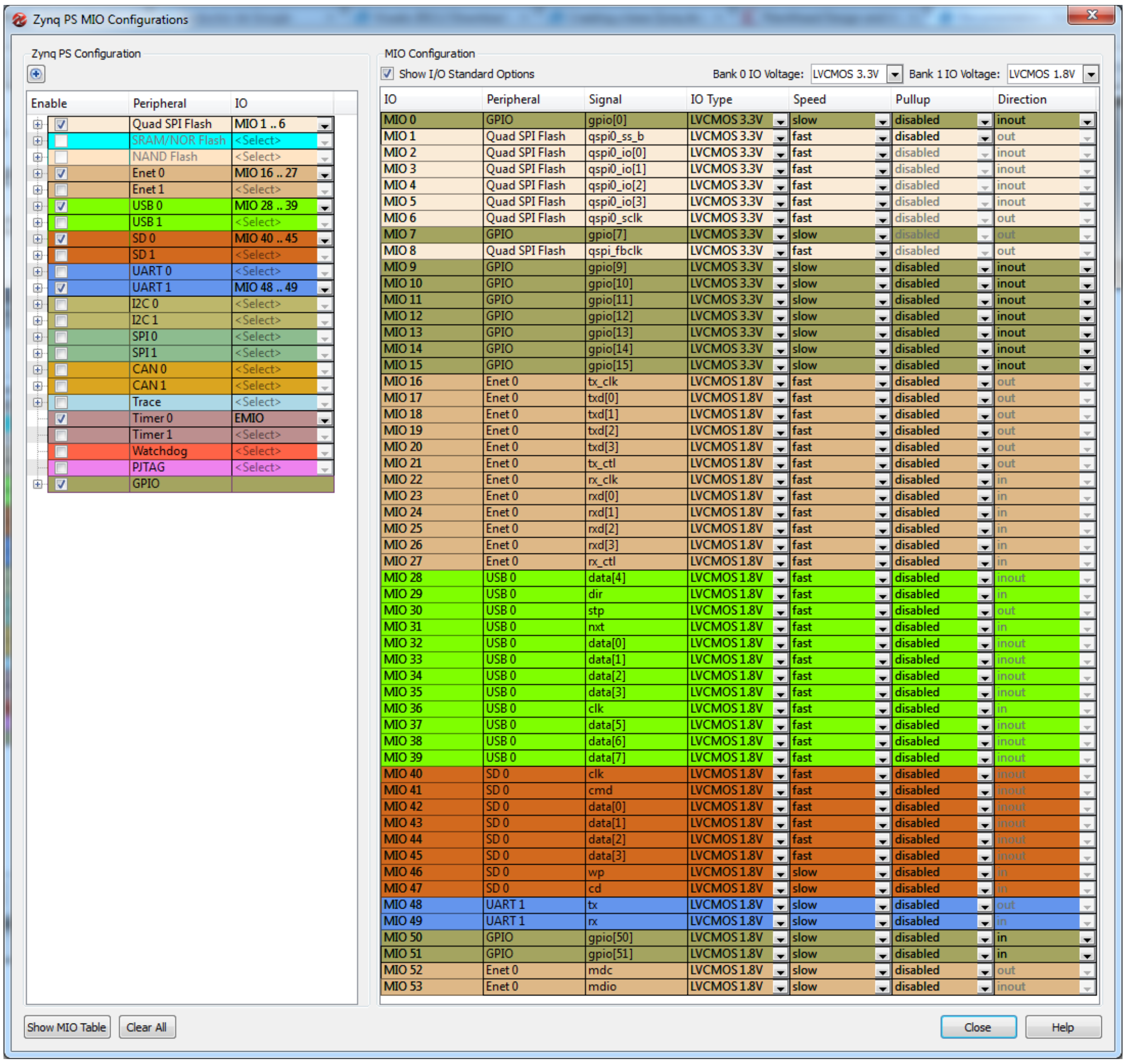
Zynq PS configuration

Peripheral	Status	Signal Group	MIO	Freq
CAN0	Disabled			
CAN1	Disabled			
ENET0	Enabled	default	MIO 16 .. 27	1000 MBPS
		GRP_MDIO	MIO 52 .. 53	
ENET1	Disabled			

Figure 78: Import PS Configuration Window 2/2

After "OK", many peripherals are now enabled in the Processing System with some MIO pins assigned to them in coordination with the ZedBoard layout. For example, UART1 is enabled and UART0 is disabled because UART1 is connected to the USB-UART bridge chip on this board.

Remember that the MIOs are listed in numerical order but the peripherals are listed from top to bottom in order of priority based on either their importance in the system (like the Flash) or how limited they are in their possible MIO mappings. Moreover, only one boot device can be selected: QSPI, NOR or NAND; SD Card is also a boot option. If Quad SPI Flash is selected NOR and NAND peripherals are grayed out because the three interfaces are mutually exclusive.


Zynq PS MIO Configurations

Zynq PS Configuration

Enable	Peripheral	IO
✓	Quad SPI Flash	MIO 1 .. 6
	SRAM/NOR Flash	<Select>
	NAND Flash	<Select>
✓	Enet 0	MIO 16 .. 27
	Enet 1	<Select>
✓	USB 0	MIO 28 .. 39
	USB 1	<Select>
✓	SD 0	MIO 40 .. 45
	SD 1	<Select>
	UART 0	<Select>
✓	UART 1	MIO 48 .. 49
	I2C 0	<Select>
	I2C 1	<Select>
	SPI 0	<Select>
	SPI 1	<Select>
	CAN 0	<Select>
	CAN 1	<Select>
	Trace	<Select>
✓	Timer 0	EMIO
	Timer 1	<Select>
	Watchdog	<Select>
	PJTAG	<Select>
✓	GPIO	

MIO Configuration

✓ Show I/O Standard Options Bank 0 IO Voltage: LVCMOS 3.3V Bank 1 IO Voltage: LVCMOS 1.8V

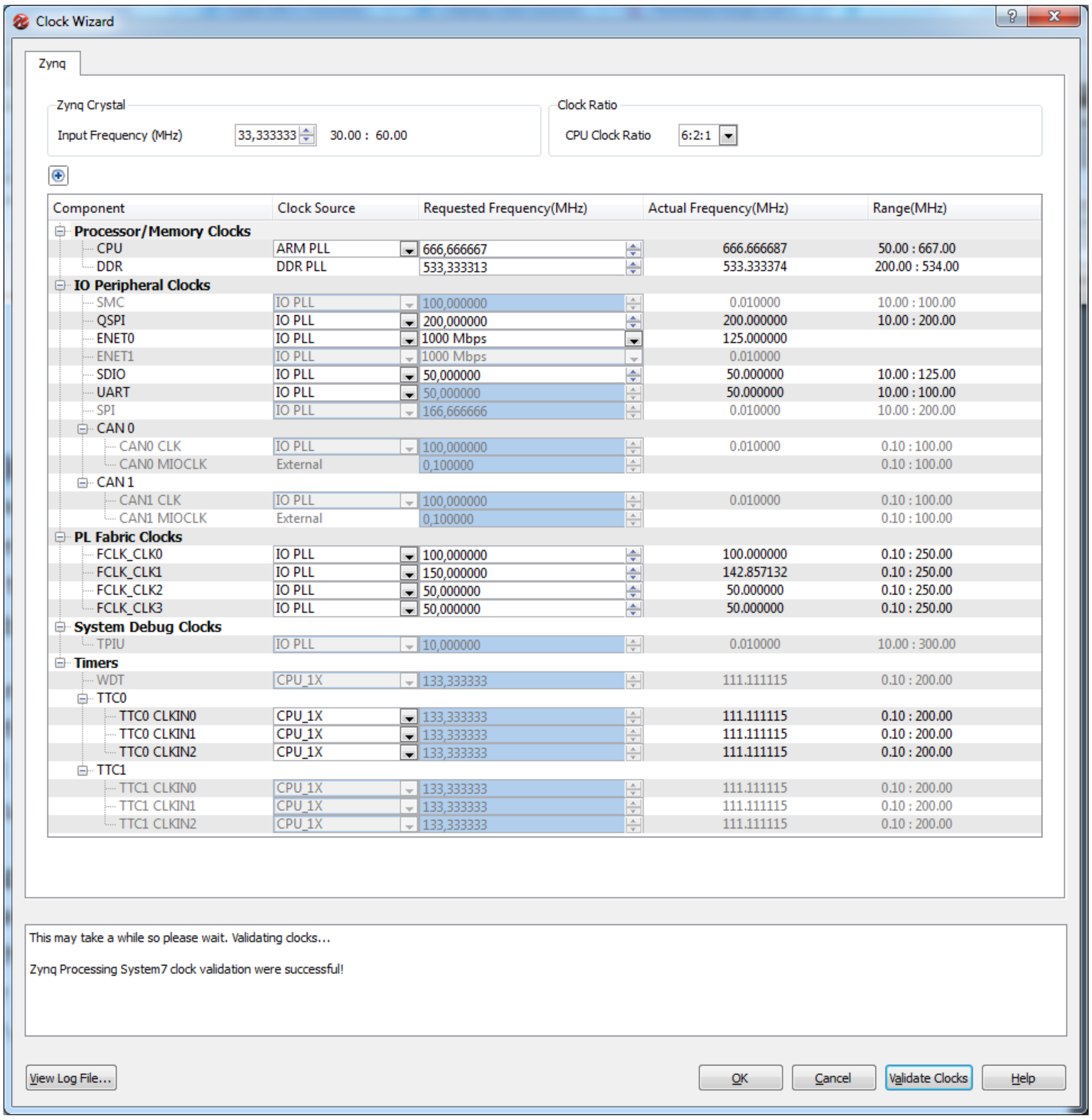
IO	Peripheral	Signal	IO Type	Speed	Pullup	Direction
MIO 0	GPIO	gpio[0]	LVCMOS 3.3V	slow	disabled	inout
MIO 1	Quad SPI Flash	qspi0_ss_b	LVCMOS 3.3V	fast	disabled	out
MIO 2	Quad SPI Flash	qspi0_io[0]	LVCMOS 3.3V	fast	disabled	inout
MIO 3	Quad SPI Flash	qspi0_io[1]	LVCMOS 3.3V	fast	disabled	inout
MIO 4	Quad SPI Flash	qspi0_io[2]	LVCMOS 3.3V	fast	disabled	inout
MIO 5	Quad SPI Flash	qspi0_io[3]	LVCMOS 3.3V	fast	disabled	inout
MIO 6	Quad SPI Flash	qspi0_sclk	LVCMOS 3.3V	fast	disabled	out
MIO 7	GPIO	gpio[7]	LVCMOS 3.3V	slow	disabled	out
MIO 8	Quad SPI Flash	qspi_fbclk	LVCMOS 3.3V	fast	disabled	out
MIO 9	GPIO	gpio[9]	LVCMOS 3.3V	slow	disabled	inout
MIO 10	GPIO	gpio[10]	LVCMOS 3.3V	slow	disabled	inout
MIO 11	GPIO	gpio[11]	LVCMOS 3.3V	slow	disabled	inout
MIO 12	GPIO	gpio[12]	LVCMOS 3.3V	slow	disabled	inout
MIO 13	GPIO	gpio[13]	LVCMOS 3.3V	slow	disabled	inout
MIO 14	GPIO	gpio[14]	LVCMOS 3.3V	slow	disabled	inout
MIO 15	GPIO	gpio[15]	LVCMOS 3.3V	slow	disabled	inout
MIO 16	Enet 0	tx_clk	LVCMOS 1.8V	fast	disabled	out
MIO 17	Enet 0	txd[0]	LVCMOS 1.8V	fast	disabled	out
MIO 18	Enet 0	txd[1]	LVCMOS 1.8V	fast	disabled	out
MIO 19	Enet 0	txd[2]	LVCMOS 1.8V	fast	disabled	out
MIO 20	Enet 0	txd[3]	LVCMOS 1.8V	fast	disabled	out
MIO 21	Enet 0	tx_ctl	LVCMOS 1.8V	fast	disabled	out
MIO 22	Enet 0	rx_clk	LVCMOS 1.8V	fast	disabled	in
MIO 23	Enet 0	rxd[0]	LVCMOS 1.8V	fast	disabled	in
MIO 24	Enet 0	rxd[1]	LVCMOS 1.8V	fast	disabled	in
MIO 25	Enet 0	rxd[2]	LVCMOS 1.8V	fast	disabled	in
MIO 26	Enet 0	rxd[3]	LVCMOS 1.8V	fast	disabled	in
MIO 27	Enet 0	rx_ctl	LVCMOS 1.8V	fast	disabled	in
MIO 28	USB 0	data[4]	LVCMOS 1.8V	fast	disabled	inout
MIO 29	USB 0	dir	LVCMOS 1.8V	fast	disabled	in
MIO 30	USB 0	stp	LVCMOS 1.8V	fast	disabled	out
MIO 31	USB 0	nxt	LVCMOS 1.8V	fast	disabled	in
MIO 32	USB 0	data[0]	LVCMOS 1.8V	fast	disabled	inout
MIO 33	USB 0	data[1]	LVCMOS 1.8V	fast	disabled	inout
MIO 34	USB 0	data[2]	LVCMOS 1.8V	fast	disabled	inout
MIO 35	USB 0	data[3]	LVCMOS 1.8V	fast	disabled	inout
MIO 36	USB 0	clk	LVCMOS 1.8V	fast	disabled	in
MIO 37	USB 0	data[5]	LVCMOS 1.8V	fast	disabled	inout
MIO 38	USB 0	data[6]	LVCMOS 1.8V	fast	disabled	inout
MIO 39	USB 0	data[7]	LVCMOS 1.8V	fast	disabled	inout
MIO 40	SD 0	clk	LVCMOS 1.8V	fast	disabled	inout
MIO 41	SD 0	cmd	LVCMOS 1.8V	fast	disabled	inout
MIO 42	SD 0	data[0]	LVCMOS 1.8V	fast	disabled	inout
MIO 43	SD 0	data[1]	LVCMOS 1.8V	fast	disabled	inout
MIO 44	SD 0	data[2]	LVCMOS 1.8V	fast	disabled	inout
MIO 45	SD 0	data[3]	LVCMOS 1.8V	fast	disabled	inout
MIO 46	SD 0	wp	LVCMOS 1.8V	slow	disabled	in
MIO 47	SD 0	cd	LVCMOS 1.8V	slow	disabled	in
MIO 48	UART 1	tx	LVCMOS 1.8V	slow	disabled	out
MIO 49	UART 1	rx	LVCMOS 1.8V	slow	disabled	in
MIO 50	GPIO	gpio[50]	LVCMOS 1.8V	slow	disabled	in
MIO 51	GPIO	gpio[51]	LVCMOS 1.8V	slow	disabled	in
MIO 52	Enet 0	mdc	LVCMOS 1.8V	slow	disabled	out
MIO 53	Enet 0	mdio	LVCMOS 1.8V	slow	disabled	inout

Figure 79: MIO Configuration

The Zynq PS MIO Configurations window can be closed and the next step is setting the PS PLL Clocks.

2.2.2.1. PS PLL Clocks

Clicking on “Clock Generation”, the Clock Wizard will be opened. In “Clock Source”, three PLL can be selected: the ARM PLL, the DDR PLL and the I/O PLL. Each uses the same input reference clock, which is 33,3333MHz on ZedBoard. Actually, any value from 30 to 60MHz is accepted, but 33.3333MHz is the standard value of the Fox clock which provides the best performance. Ensure that the final result is the same and this window can be close.



Zynq Crystal
Input Frequency (MHz) 33,333333 30.00 : 60.00

Clock Ratio
CPU Clock Ratio 6:2:1

Component	Clock Source	Requested Frequency(MHz)	Actual Frequency(MHz)	Range(MHz)
Processor/Memory Clocks				
CPU	ARM PLL	666,666667	666.666687	50.00 : 667.00
DDR	DDR PLL	533,333313	533.333374	200.00 : 534.00
IO Peripheral Clocks				
SMC	IO PLL	100,000000	0.010000	10.00 : 100.00
QSPI	IO PLL	200,000000	200.000000	10.00 : 200.00
ENET0	IO PLL	1000 Mbps	125.000000	
ENET1	IO PLL	1000 Mbps	0.010000	
SDIO	IO PLL	50,000000	50.000000	10.00 : 125.00
UART	IO PLL	50,000000	50.000000	10.00 : 100.00
SPI	IO PLL	166,666666	0.010000	10.00 : 200.00
CAN 0				
CAN0 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN0 MIOCLK	External	0,100000		0.10 : 100.00
CAN 1				
CAN1 CLK	IO PLL	100,000000	0.010000	0.10 : 100.00
CAN1 MIOCLK	External	0,100000		0.10 : 100.00
PL Fabric Clocks				
FCLK_CLK0	IO PLL	100,000000	100.000000	0.10 : 250.00
FCLK_CLK1	IO PLL	150,000000	142.857132	0.10 : 250.00
FCLK_CLK2	IO PLL	50,000000	50.000000	0.10 : 250.00
FCLK_CLK3	IO PLL	50,000000	50.000000	0.10 : 250.00
System Debug Clocks				
TPIU	IO PLL	10,000000	0.010000	10.00 : 300.00
Timers				
WDT	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0				
TTC0 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC0 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1				
TTC1 CLKIN0	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN1	CPU_1X	133,333333	111.111115	0.10 : 200.00
TTC1 CLKIN2	CPU_1X	133,333333	111.111115	0.10 : 200.00

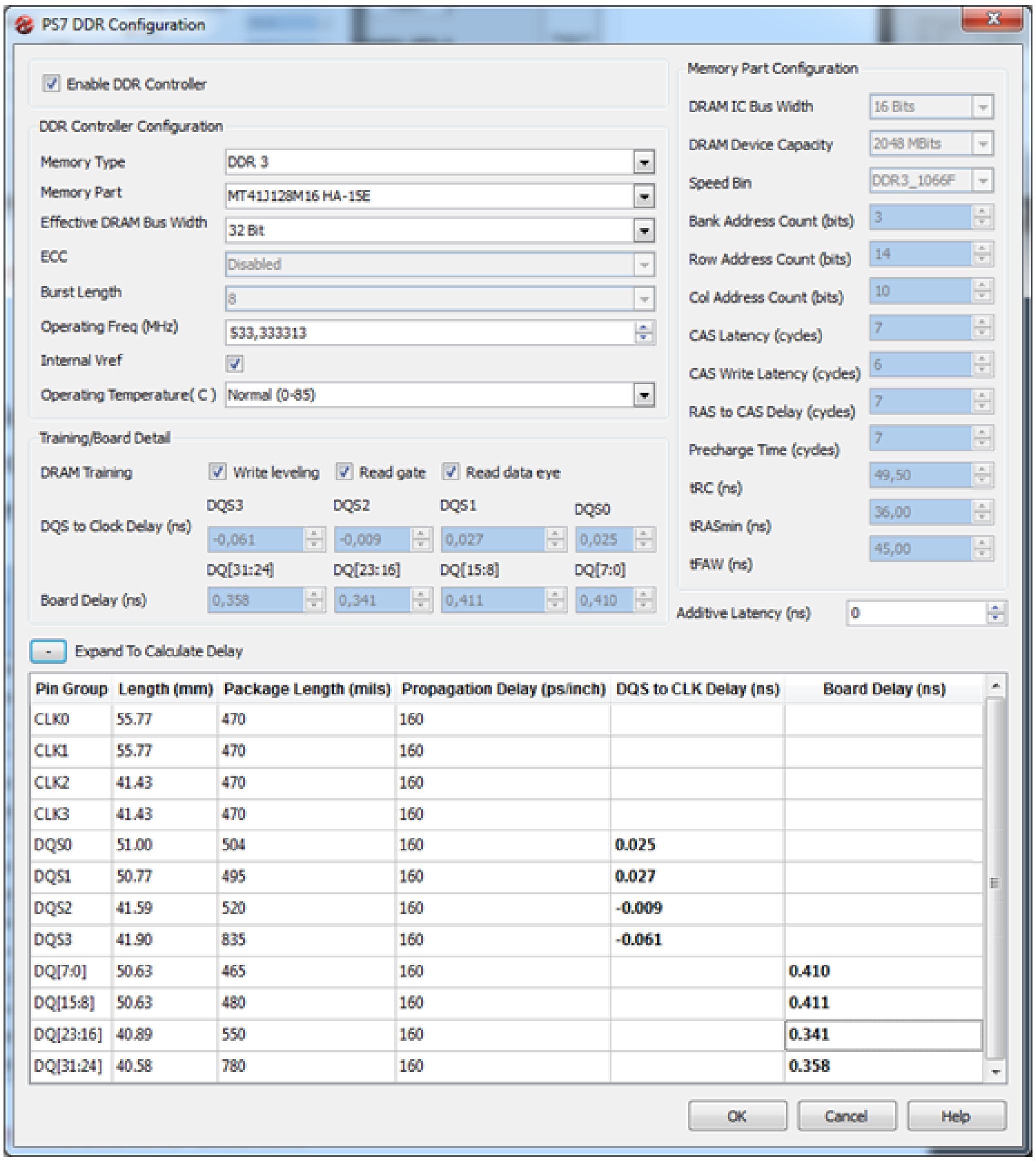
```
This may take a while so please wait. Validating clocks...

Zynq Processing System7 clock validation were successful!
```

Figure 80: Clock Wizard

2.2.2.2. DDR3 Memory

Clicking on “Memory Interfaces”, the PS7 DDR Configuration screen is showed allowing the configuration of the DDR Controller, the Memory Part and the board details used for DDR interface. For best DDR3 performance, DRAM training must be enabled for write leveling, read gate, and read data eye options in the PS Configuration Tool in Xilinx Platform Studio (XPS).

The ZedBoard configuration was previously exported, so the configuration is almost completed. "Expand To Calculate Delay" must be clicked and the columns "Length (mm)" and "Package Length (mils)" must be filled with the following details found on the ZedBoard Hardware User Guide. After filling the data, the result is the following image.


PS7 DDR Configuration

[✓] Enable DDR Controller

DDR Controller Configuration

Memory Type	DDR 3
Memory Part	MT41J128M16 HA-15E
Effective DRAM Bus Width	32 Bit
ECC	Disabled
Burst Length	8
Operating Freq (MHz)	533,333313
Internal Vref	✓
Operating Temperature(C)	Normal (0-85)

Training/Board Detail

DRAM Training: [✓] Write leveling [✓] Read gate [✓] Read data eye

	DQS3	DQS2	DQS1	DQS0
DQS to Clock Delay (ns)	-0,061	-0,009	0,027	0,025
	DQ[31:24]	DQ[23:16]	DQ[15:8]	DQ[7:0]
Board Delay (ns)	0,358	0,341	0,411	0,410

Memory Part Configuration

DRAM IC Bus Width	16 Bits
DRAM Device Capacity	2048 MBits
Speed Bin	DDR3_1066F
Bank Address Count (bits)	3
Row Address Count (bits)	14
Col Address Count (bits)	10
CAS Latency (cycles)	7
CAS Write Latency (cycles)	6
RAS to CAS Delay (cycles)	7
Precharge Time (cycles)	7
tRC (ns)	49,50
tRASmin (ns)	36,00
tFAW (ns)	45,00
Additive Latency (ns)	0

Expand To Calculate Delay

Pin Group	Length (mm)	Package Length (mils)	Propagation Delay (ps/inch)	DQS to CLK Delay (ns)	Board Delay (ns)
CLK0	55.77	470	160		
CLK1	55.77	470	160		
CLK2	41.43	470	160		
CLK3	41.43	470	160		
DQS0	51.00	504	160	0.025	
DQS1	50.77	495	160	0.027	
DQS2	41.59	520	160	-0.009	
DQS3	41.90	835	160	-0.061	
DQ[7:0]	50.63	465	160		0.410
DQ[15:8]	50.63	480	160		0.411
DQ[23:16]	40.89	550	160		0.341
DQ[31:24]	40.58	780	160		0.358

Figure 81: DDR Configuration

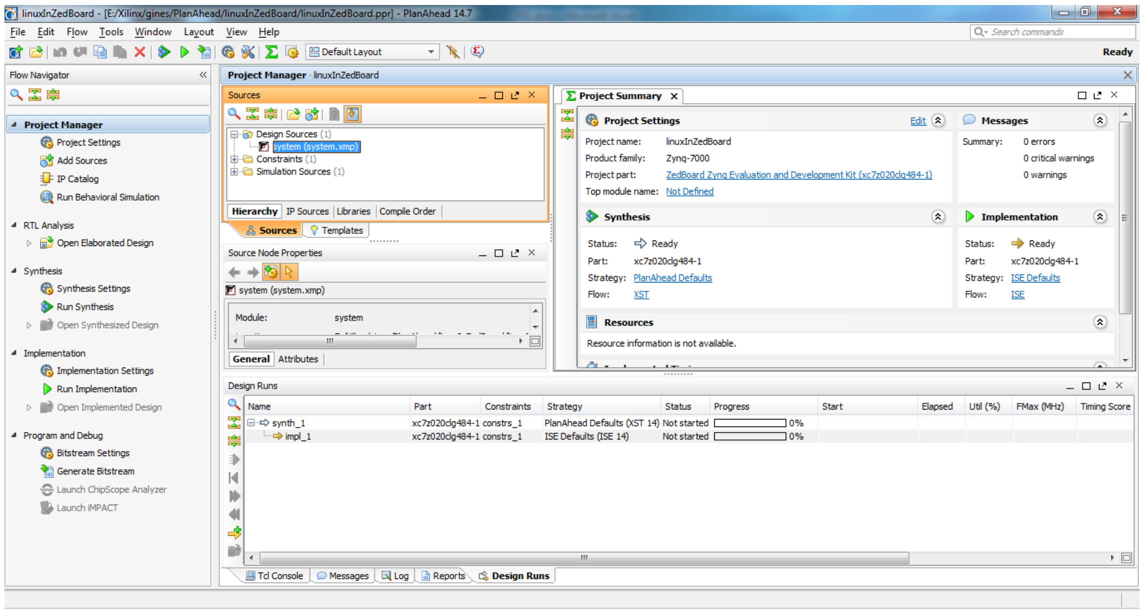
The XPS tool is no longer necessary, it can be closed. A basic ARM hardware platform is now configured. The configuration includes clock and DDR controller settings.

2.3.2. PlanAhead

Now hardware platform will be built and exported to the Software Development Kit (SDK) so that an application can be developed. Once XPS is closed, the top-level project design file system.xmp and the Microprocessors Hardware Specification file *.mhs are added in the Sources view in the PlanAhead tool.

If the Embedded Design Sources is expanded, the target files associated with the sub-design will be showed (only "system.xmp" in this case). Moreover, other files are created, for instance "ps7_init.c", "ps7_init.h", "ps7_init.td" and "ps7_init.html".

The program looks like as following:



Module: system

Project Settings

Project name: linuxInZedBoard
Product family: Zynq-7000
Project part: ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name: Not Defined

Messages

Summary: 0 errors
0 critical warnings
0 warnings

Synthesis

Status: Ready
Part: xc7z020clg484-1
Strategy: PlanAhead Defaults
Flow: XST

Implementation

Status: Ready
Part: xc7z020clg484-1
Strategy: ISE Defaults
Flow: ISE

Resources

Resource information is not available.

Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timing Score
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	Not started	0%					
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	Not started	0%					

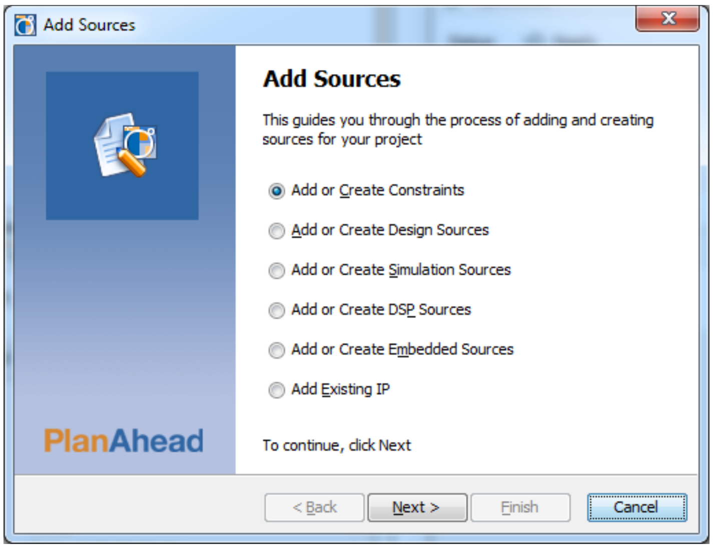
Figure 82: PlanAhead after Adding "system.xmp"

2.3. Design Constraints

The PL needs a User Constraint File (UCF) to define the pin locations and PS timing. The ZedBoard *.ucf file (zedboard_master_UCF_RevC_v3.ucf) can be downloaded on ZedBoard official webpage and imported such as the above-mentioned PS7 configuration file.

2.3.1. PlanAhead

In "Project Manager", click in "Add Sources" and "Add or Create Constraints":



Add Sources

This guides you through the process of adding and creating sources for your project

[✓ Add or Create Constraints]
Add or Create Design Sources
Add or Create Simulation Sources
Add or Create DSP Sources
Add or Create Embedded Sources
Add Existing IP

To continue, click Next

Figure 83: Add Sources Window 2/2

Now, click in "Add Files", choose "zedboard_master_UCF_RevC_v3.ucf" file and "Finish":

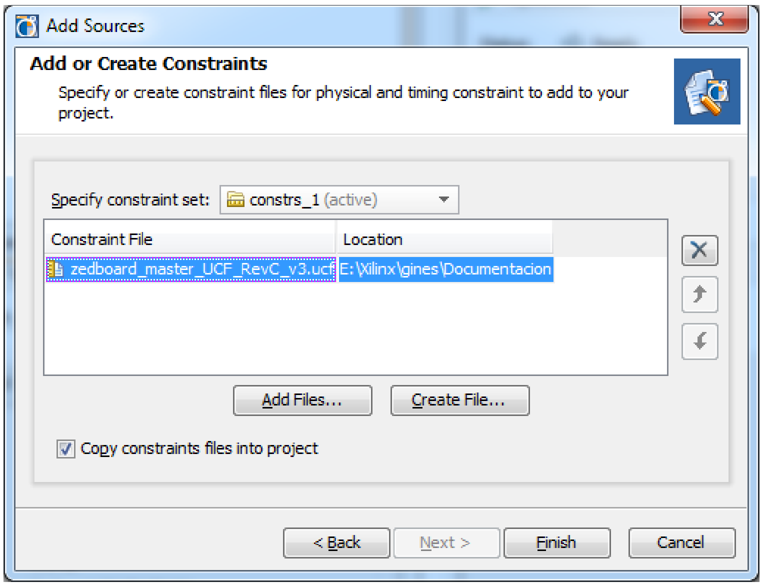

Add Sources
Add or Create Constraints
Specify or create constraint files for physical and timing constraint to add to your project.
Specify constraint set:
constrs_1 (active)
Constraint File
Location
zedboard_master_UCF_RevC_v3.ucf
E:\Xilinx\gines\Documentacion
Add Files...
Create File...
Copy constraints files into project
< Back
Next >
Finish
Cancel

Figure 84: Add Constraints Window

The "Sources" Window must looks like the image below:

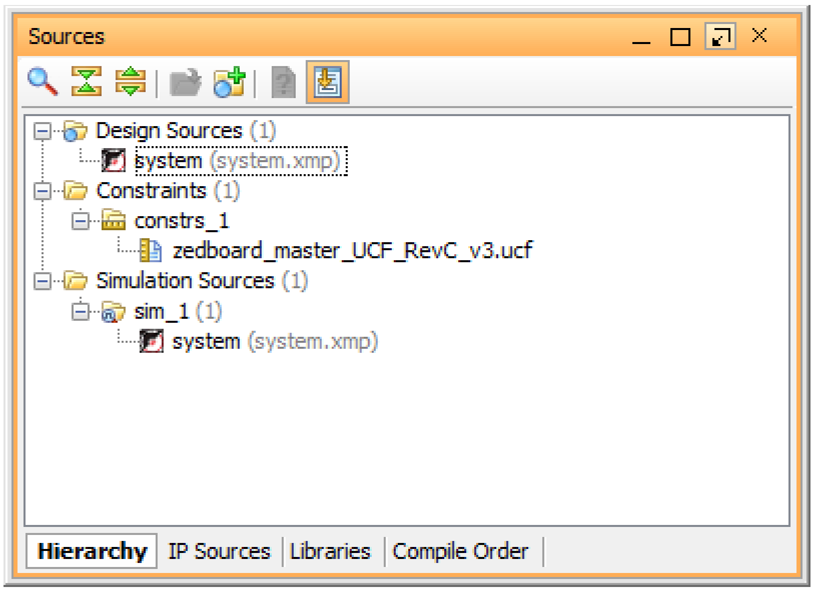
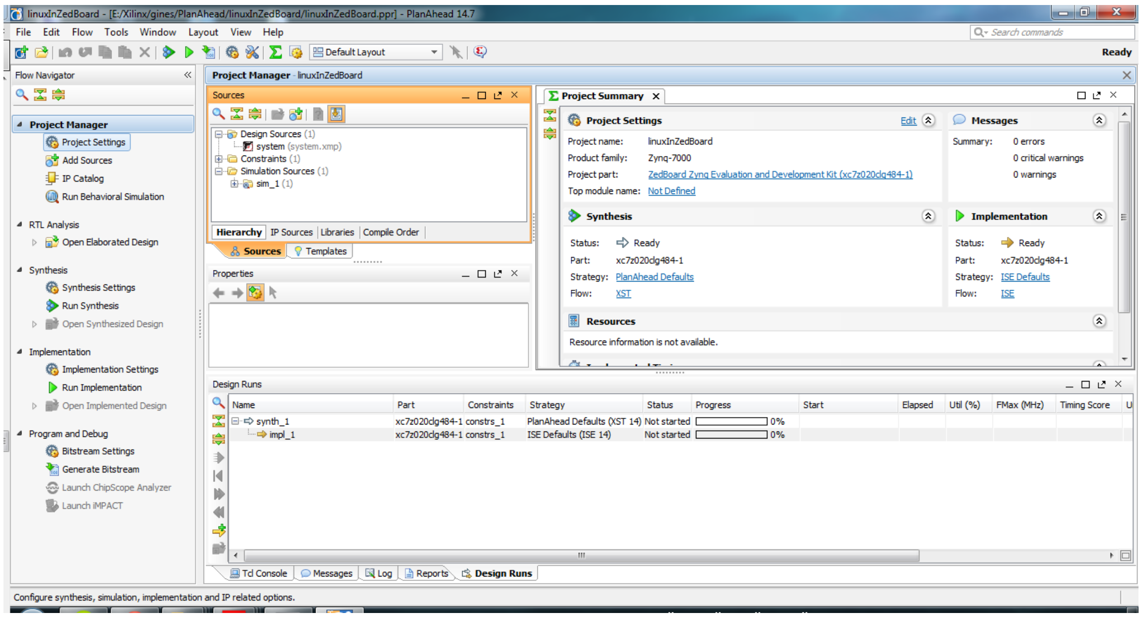


Figure 85: Sources Window after Adding the Constraints File



Project Settings Edit

Project name:	linuxInZedBoard
Product family:	Zynq-7000
Project part:	ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name:	Not Defined

Messages

Summary: 0 errors
0 critical warnings
0 warnings

Synthesis

Status:	Ready
Part:	xc7z020clg484-1
Strategy:	PlanAhead Defaults
Flow:	XST

Implementation

Status:	Ready
Part:	xc7z020clg484-1
Strategy:	ISE Defaults
Flow:	ISE

Resources

Resource information is not available.

Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timing Score
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	Not started	0%					
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	Not started	0%					

Configure synthesis, simulation, implementation and IP related options.

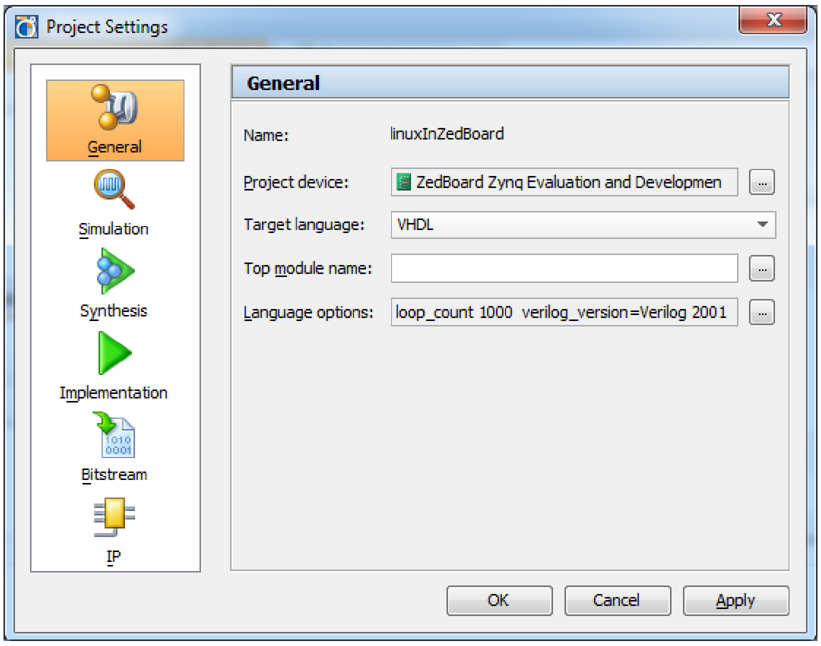
Figure 86: PlanAhead after Adding the Constraints File

2.4. Top HDL and Hardware Platform Building

The hardware platform has to be built and exported to SDK for being able to develop any application. Building includes the synthesis, the implementation and the BitStream generation.

2.4.1. PlanAhead

Before creating the Top HDL, the Project Settings should be checked. They are in the "Project Manager" Panel. The "Target language" must be fixed as "VHDL" and "OK":



General

Name:	linuxInZedBoard
Project device:	ZedBoard Zynq Evaluation and Developmen
Target language:	VHDL
Top module name:	
Language options:	loop_count 1000 verilog_version=Verilog 2001

Figure 87: Project Settings

"Top module name" will look empty until the Top HDL be created. The top level wrapper for the design will be created by right clicking in "system (system.xmp)" and "Create Top HDL":

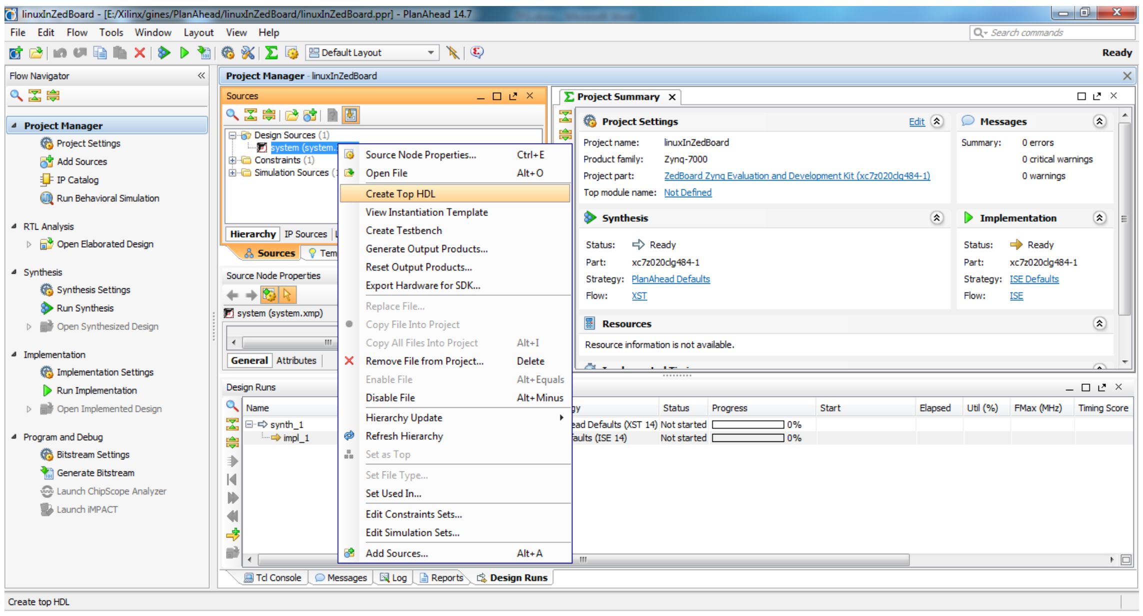

linuxInZedBoard - [E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.ppr] - PlanAhead 14.7
Source Node Properties... Ctrl+E
Open File Alt+O
Create Top HDL
View Instantiation Template
Create Testbench
Generate Output Products...
Reset Output Products...
Export Hardware for SDK...
Replace File...
Copy File Into Project
Copy All Files Into Project Alt+I
Remove File from Project... Delete
Enable File Alt+Equals
Disable File Alt+Minus
Hierarchy Update
Refresh Hierarchy
Set as Top
Set File Type...
Set Used In...
Edit Constraints Sets...
Edit Simulation Sets...
Add Sources... Alt+A
Project name: linuxInZedBoard
Product family: Zynq-7000
Project part: ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)
Top module name: Not Defined
Create top HDL

Figure 88: Create Top HDL Option

PlanAhead generates a "system_stub.vhd" top-level module for the design where "system.xmp" is now a sub-module of system_stub.

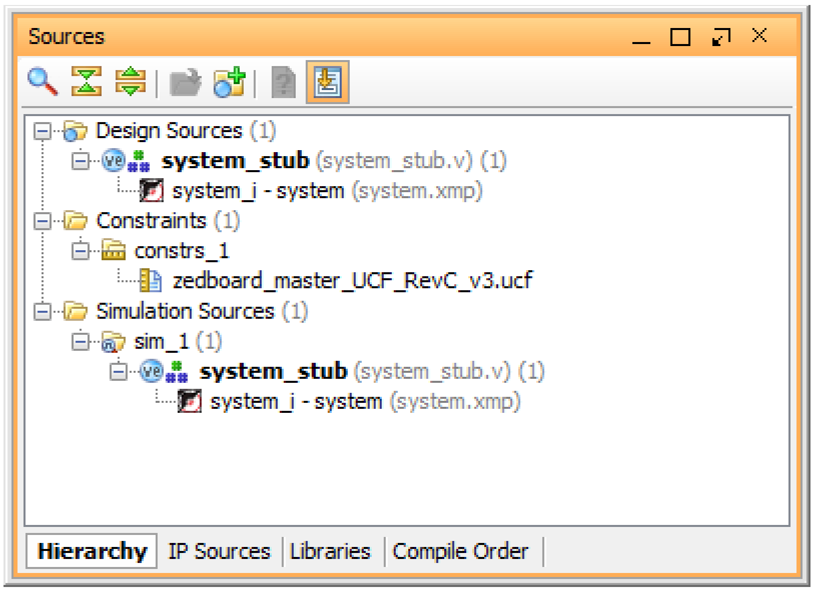
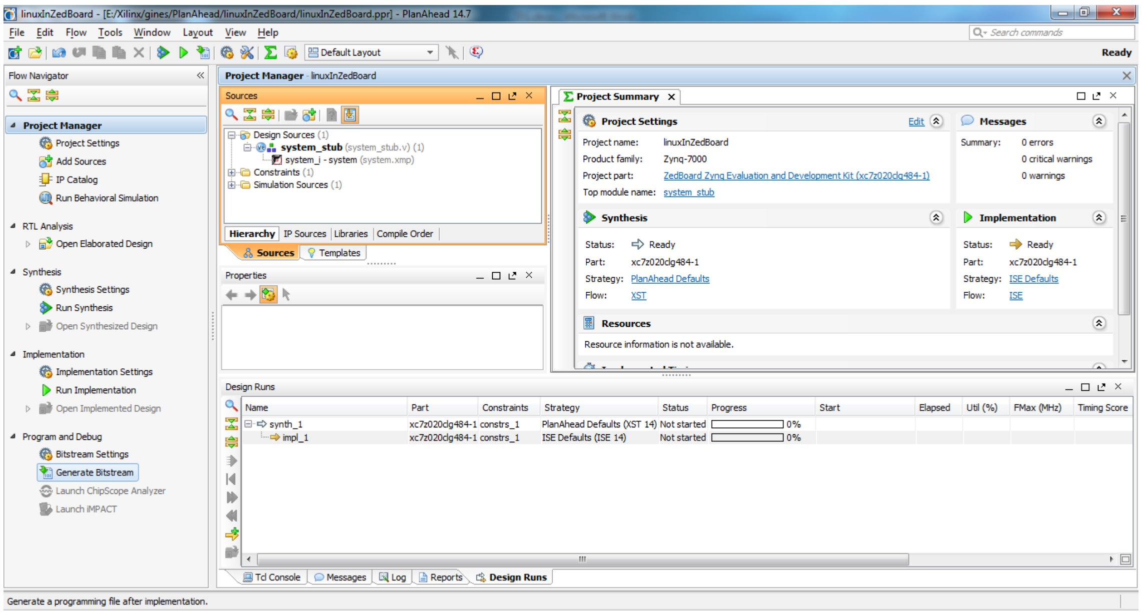

Sources
Design Sources (1)
system_stub (system_stub.v) (1)
system_i - system (system.xmp)
Constraints (1)
constrs_1
zedboard_master_UCF_RevC_v3.ucf
Simulation Sources (1)
sim_1 (1)
system_stub (system_stub.v) (1)
system_i - system (system.xmp)
Hierarchy
IP Sources
Libraries
Compile Order

Figure 89: Sources Window after Adding the Top HDL

The next processes which are needed are the synthesis, implementation, verification and BitStream generation. In the left panel of PlanAhead there are the main buttons to configure and launch them. Nevertheless, clicking in "Generate BitStream" will launch automatically the entire process:



Project Settings

Project name: linuxInZedBoard

Product family: Zynq-7000

Project part: ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)

Top module name: system_stub

Messages

Summary: 0 errors, 0 critical warnings, 0 warnings

Synthesis

Status: Ready

Part: xc7z020clg484-1

Strategy: PlanAhead Defaults

Flow: XST

Implementation

Status: Ready

Part: xc7z020clg484-1

Strategy: ISE Defaults

Flow: ISE

Resources

Resource information is not available.

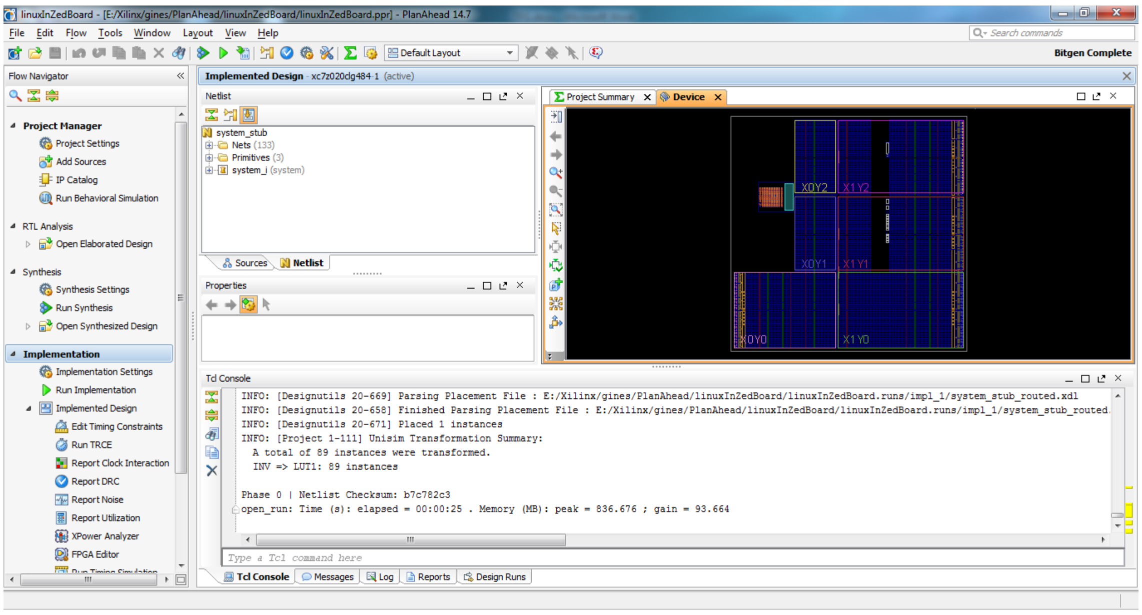
Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timing Score
synth_1	xc7z020clg484-1	constrs_1	PlanAhead Defaults (XST 14)	Not started	0%					
impl_1	xc7z020clg484-1	constrs_1	ISE Defaults (ISE 14)	Not started	0%					

Generate a programming file after implementation.

Figure 90: PlanAhead after Adding the Top HDL

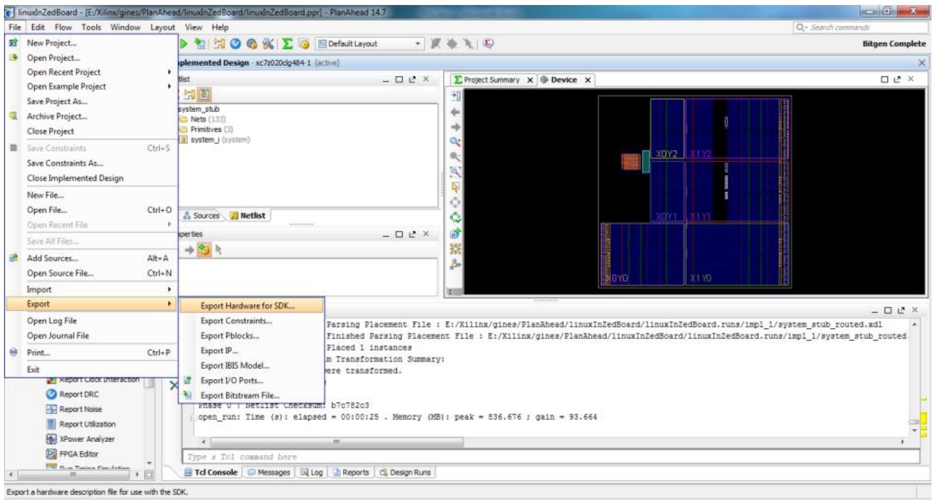
The BitStream finishes the hardware design. There are not errors, only some warnings, so the warnings/errors windows can be closed by clicking “OK”.



```
INFO: [Designutils 20-669] Parsing Placement File : E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.runs/impl_1/system_stub_routed.xdl
INFO: [Designutils 20-658] Finished Parsing Placement File : E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.runs/impl_1/system_stub_routed.
INFO: [Designutils 20-671] Placed 1 instances
INFO: [Project 1-111] Unisim Transformation Summary:
  A total of 89 instances were transformed.
  INV => LUT1: 89 instances

Phase 0 | Netlist Checksum: b7c782c3
open_run: Time (s): elapsed = 00:00:25 . Memory (MB): peak = 836.676 ; gain = 93.664
```

Figure 91: PlanAhead after Generating the BitStream File

Software project associated with the hardware design has to be created within SDK. For this purpose, the project will be exported to SDK. Click in “File”, “Export” and “Export Hardware to SDK…”:

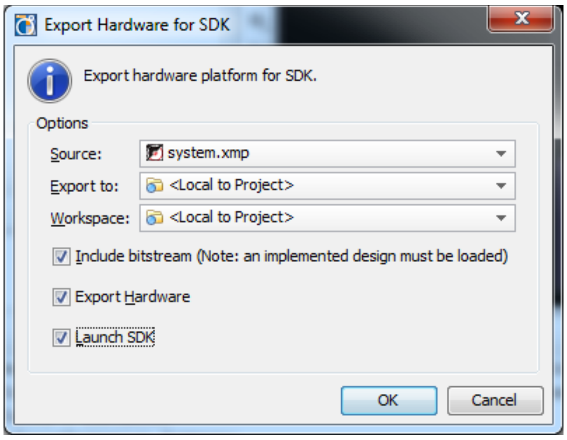


```
Parsing Placement File : E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.runs/impl_1/system_stub_routed.xdl
Finished Parsing Placement File : E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.runs/impl_1/system_stub_routed
Placed 1 instances
m Transformation Summary:
ere transformed.
Phase 0 | Netlist Checksum: b7c782c3
open_run: Time (s): elapsed = 00:00:25 . Memory (MB): peak = 836.676 ; gain = 93.664
```

Export a hardware description file for use with the SDK.

Figure 92: Export Hardware for SDK Option

Select "Include BitStream", "Export Hardware" and "Launch SDK", click in "OK":



Export hardware platform for SDK.

Options

Source: system.xmp

Export to: <Local to Project>

Workspace: <Local to Project>

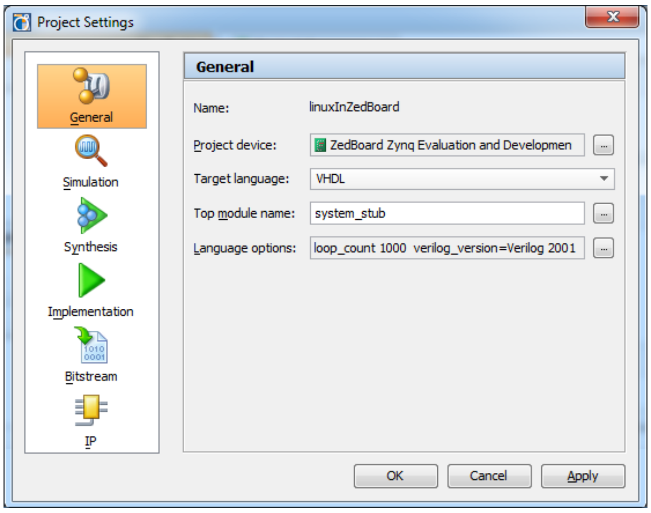
[✓] Include bitstream (Note: an implemented design must be loaded)

[✓] Export Hardware

[✓] Launch SDK

Figure 93: Export Hardware Settings

If "Project Settings" is displayed again, the "Top module name" section will not be empty this time.



General

Name: linuxInZedBoard

Project device: ZedBoard Zynq Evaluation and Developmen

Target language: VHDL

Top module name: system_stub

Language options: loop_count 1000 verilog_version=Verilog 2001

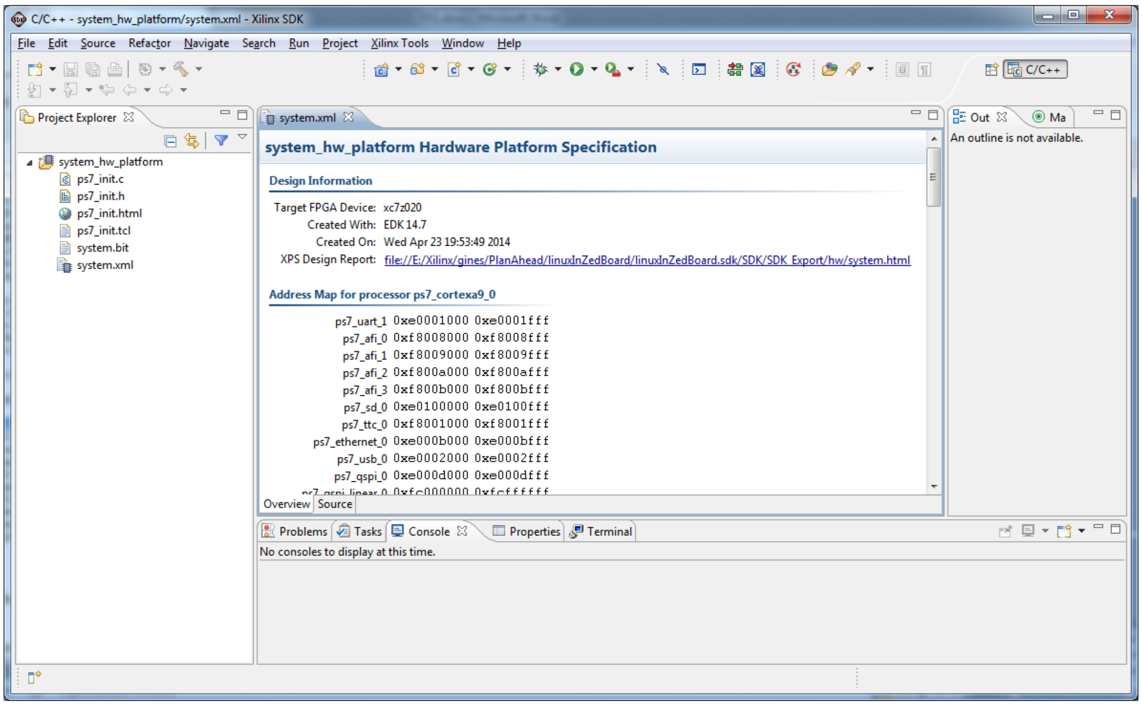
Figure 94: Project Settings after Adding the Top HDL

After this process finishes, SDK will be launched. PlanAhead can be closed; it is not further needed in this guide.

2.4.2. Software Development Kit (SDK)

The Hardware Platform Specification “system.xml” has been exported by PlanAhead to SDK. Once SDK has been launched, the “system.xml” file should be opened. It contains the memory map and associated IP blocks for each of the hardware peripherals that were connected to the processing system in XPS. To open this file at other times, double-click on “system.xml” in Project Explorer panel under the “system_hw_platform” project.

Moreover, remember that “ps7_init.c”, “ps7_init.h”, “ps7_init.tcl” and “ps7_init.html” have been also exported. At this point, SDK will look like the following image:



system_hw_platform Hardware Platform Specification

Design Information

Target FPGA Device: xc7z020
Created With: EDK 14.7
Created On: Wed Apr 23 19:53:49 2014
XPS Design Report: file://E:/Xilinx/gines/PlanAhead/linuxInZedBoard/linuxInZedBoard.sdk/SDK/SDK_Export/hw/system.html

Address Map for processor ps7_cortexa9_0

```
          ps7_uart_1 0xe0001000 0xe0001fff
           ps7_afi_0 0xf8008000 0xf8008fff
           ps7_afi_1 0xf8009000 0xf8009fff
           ps7_afi_2 0xf800a000 0xf800afff
           ps7_afi_3 0xf800b000 0xf800bfff
            ps7_sd_0 0xe0100000 0xe0100fff
           ps7_ttc_0 0xf8001000 0xf8001fff
      ps7_ethernet_0 0xe000b000 0xe000bfff
           ps7_usb_0 0xe0002000 0xe0002fff
          ps7_qspi_0 0xe000d000 0xe000dfff
   ps7_qspi_linear_0 0xfc000000 0xfcffffff
```

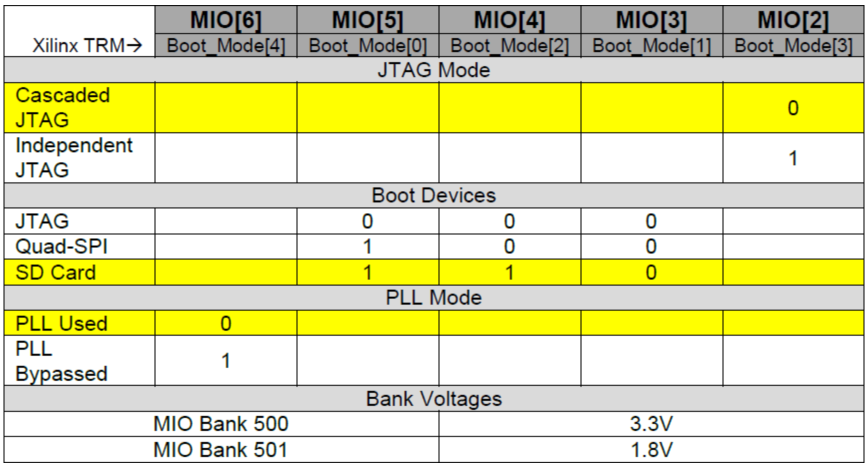
No consoles to display at this time.

Figure 95: Project Exported to SDK

Once the hardware platform is successfully exported, the first boot loader will be explained and created in the next subchapter.

2.5. Stage 0 or BootROM

The Linux OS will be introduced on ZedBoard using the SD Card, hence the MIO Configuration Modes are the list below, with the required setting highlighted in yellow:



Xilinx TRM→	MIO[6] Boot_Mode[4]	MIO[5] Boot_Mode[0]	MIO[4] Boot_Mode[2]	MIO[3] Boot_Mode[1]	MIO[2] Boot_Mode[3]
JTAG Mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Mode					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

Figure 96: ZedBoard Configuration Modes

Therefore, the jumpers must be fixed as the following image:

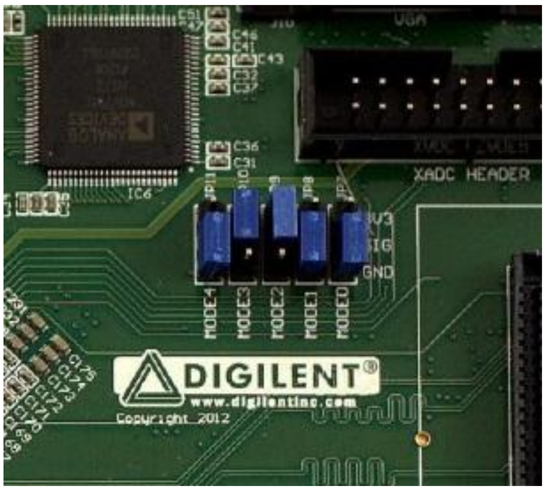


Figure 97: ZedBoard SD Card Boot Mode Jumper Setting

2.6. Boot Image, "boot.bin"

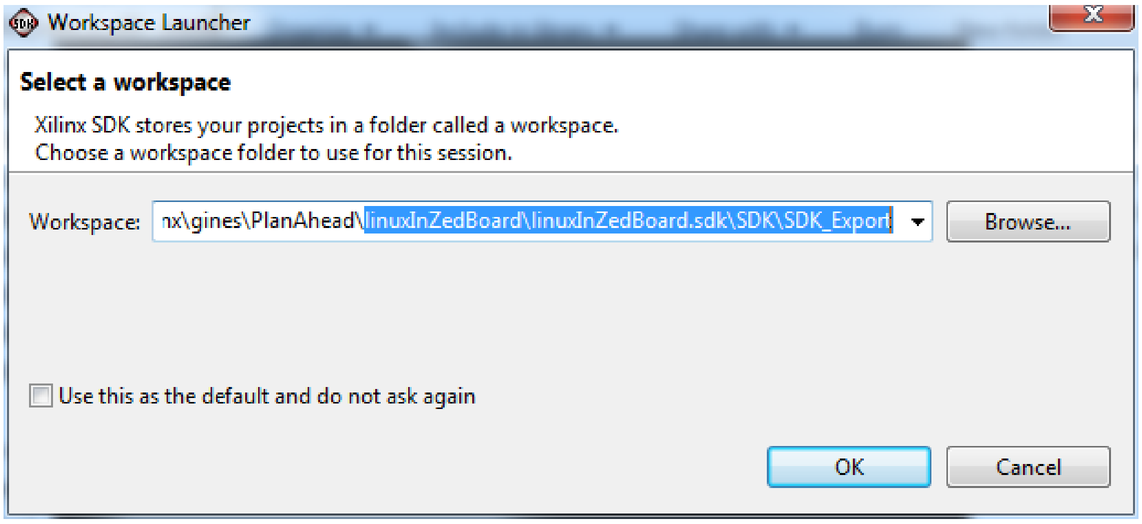
The FSBL supports multiple partitions, and each partition can be a code image or a BitStream. In this case, it will be only consisted on 3 files:

- The specific First Stage Boot Loader (FSBL) for ZedBoard.
- The specific Programmable Logic Hardware BitStream file for ZedBoard (optional).
- The U-Boot file, as Second Stage Boot Loader.

2.6.1. First Stage Boot Loader (FSBL) and BitStream

2.6.1.1. Software Development Kit (SDK)

If SDK was closed, it will be opened again with the appropriate path:



Workspace Launcher

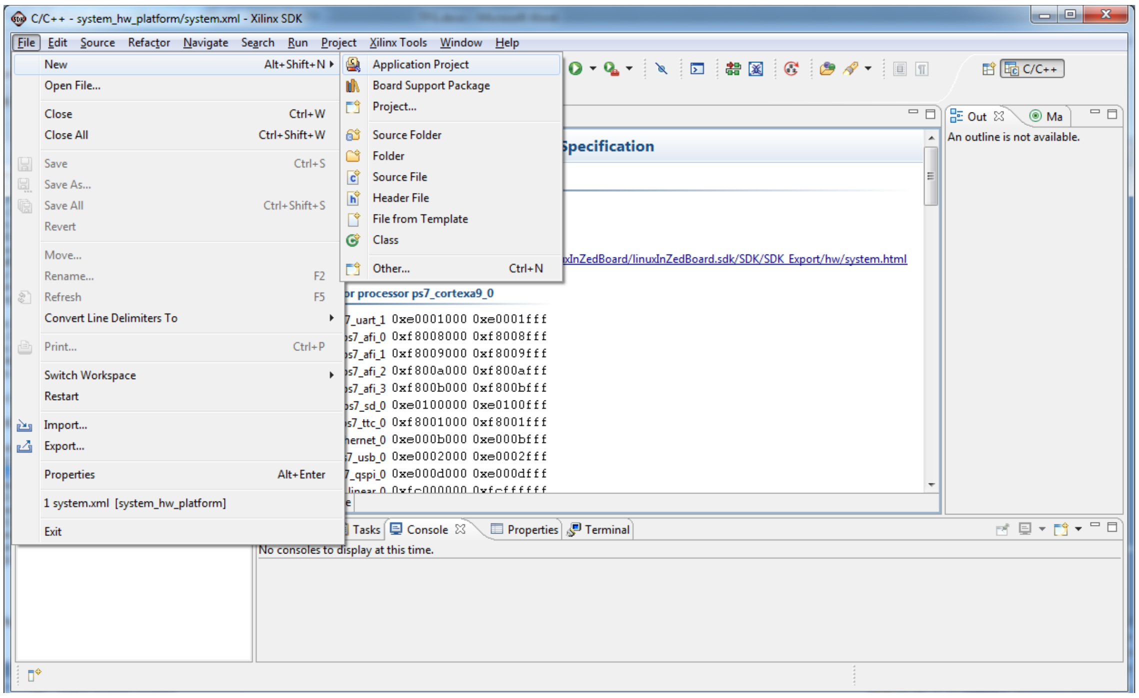
Select a workspace

Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: nx\gines\PlanAhead\linuxInZedBoard\linuxInZedBoard.sdk\SDK\SDK_Export

Figure 98: Path Selection in SDK

Once SDK is anew launched, click on “File”, “New” and “Application Project”.

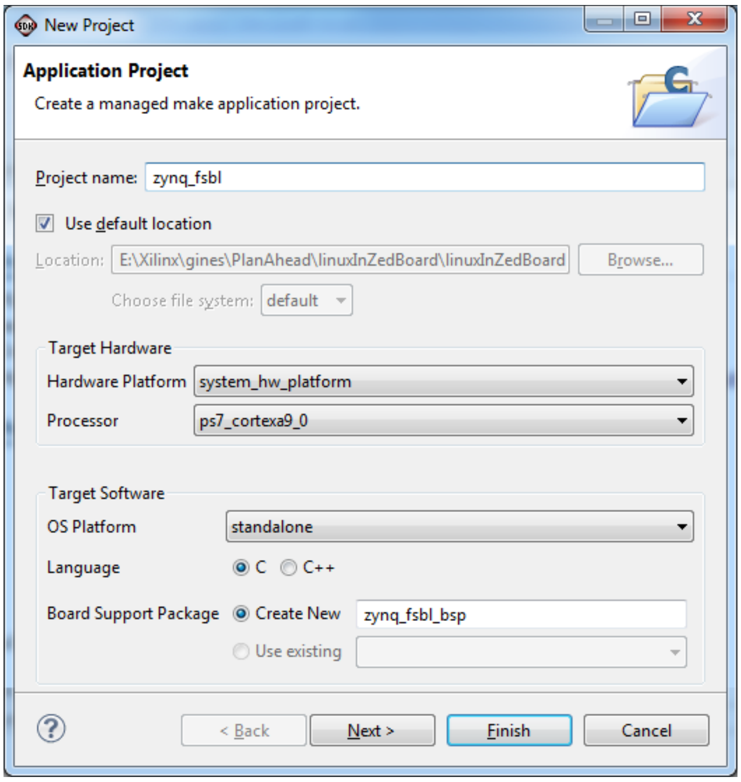


```
7_uart_1  0xe0001000 0xe0001fff
s7_afi_0  0xf8008000 0xf8008fff
s7_afi_1  0xf8009000 0xf8009fff
s7_afi_2  0xf800a000 0xf800afff
s7_afi_3  0xf800b000 0xf800bfff
s7_sd_0   0xe0100000 0xe0100fff
s7_ttc_0  0xf8001000 0xf8001fff
hernet_0  0xe000b000 0xe000bfff
7_usb_0   0xe0002000 0xe0002fff
7_qspi_0  0xe000d000 0xe000dfff
```

Figure 99: Creating an Application Project Option

After that, the application project wizard will pop up. A project name has to be written, for instance, “zynq_fsbl” and the OS Platform will be “standalone”. After that, click “Next.

Remember that the OS Platform must be selected as "standalone" instead of "linux", because the FSBL is always related to the board, regardless of whether an OS is used. Furthermore, the Hardware Platform will be automatically fixed in "system_hw_platform" and the processor in "ps7_cortexa9_0".



Project name: zynq_fsbl

[✓] Use default location

Location: E:\Xilinx\gines\PlanAhead\linuxInZedBoard\linuxInZedBoard

Choose file system: default

Target Hardware

Hardware Platform: system_hw_platform

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

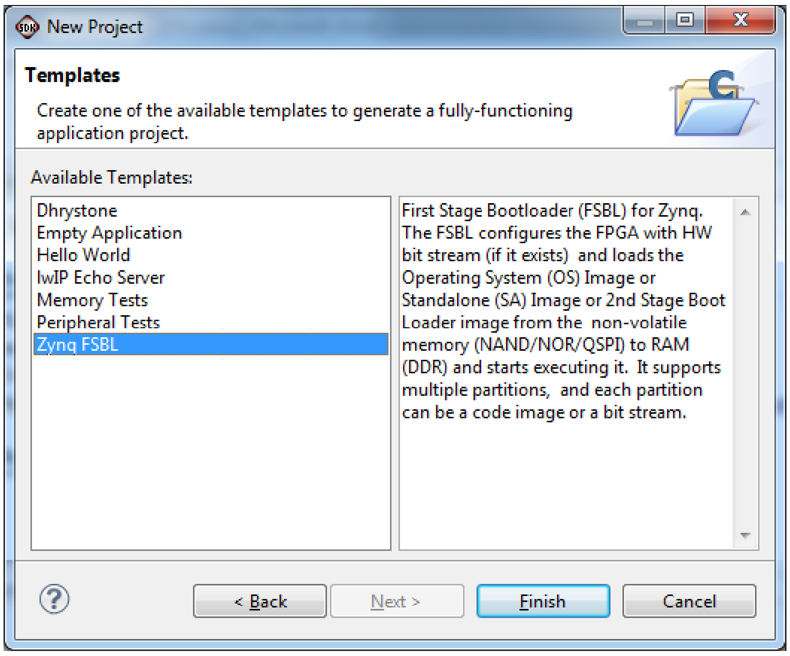
Language: [✓ C] C++

Board Support Package: [✓ Create New] zynq_fsbl_bsp

Use existing

Figure 100: New Project Settings

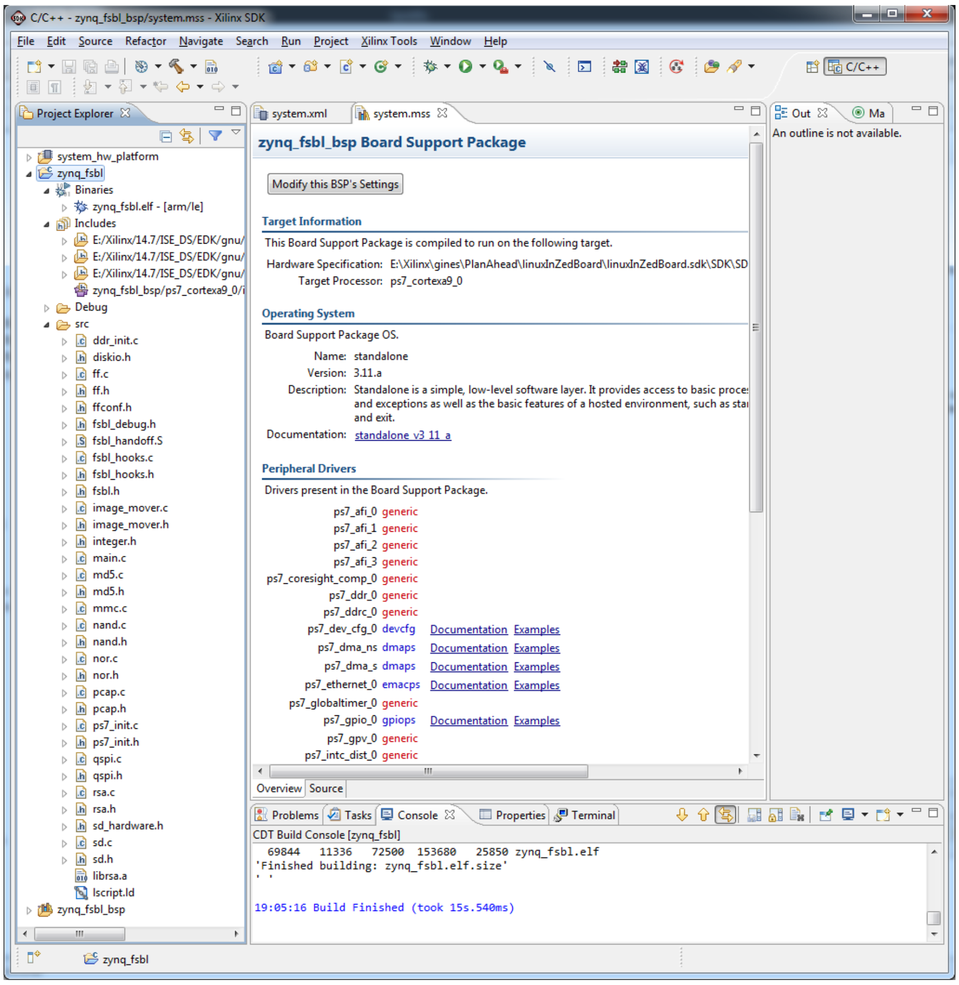
In the next window, "Zynq FSBL" has to be selected and "Finish":



First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Figure 101: Creating a Zynq FSBL

The FSBL is now created within the same workspace and automatically built by the compiler:



zynq_fsbl_bsp Board Support Package

Target Information

This Board Support Package is compiled to run on the following target.

Hardware Specification: E:\Xilinx\gines\PlanAhead\linuxInZedBoard\linuxInZedBoard.sdk\SDK\SD

Target Processor: ps7_cortexa9_0

Operating System

Board Support Package OS.

Name: standalone

Version: 3.11.a

Description: Standalone is a simple, low-level software layer. It provides access to basic proce and exceptions as well as the basic features of a hosted environment, such as sta and exit.

Documentation: standalone v3 11 a

Peripheral Drivers

Drivers present in the Board Support Package.

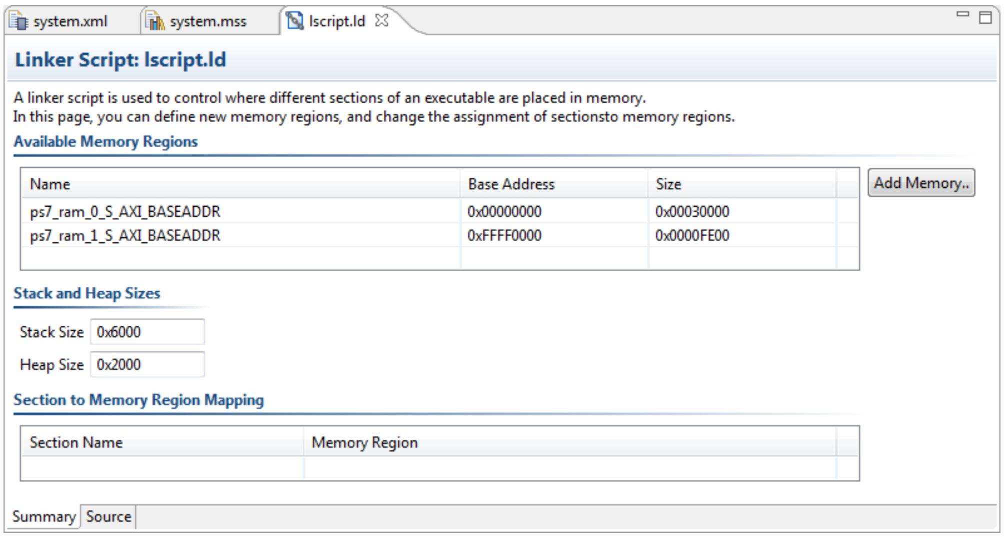
ps7_afi_0	generic		
ps7_afi_1	generic		
ps7_afi_2	generic		
ps7_afi_3	generic		
ps7_coresight_comp_0	generic		
ps7_ddr_0	generic		
ps7_ddrc_0	generic		
ps7_dev_cfg_0	devcfg	Documentation	Examples
ps7_dma_ns	dmaps	Documentation	Examples
ps7_dma_s	dmaps	Documentation	Examples
ps7_ethernet_0	emacps	Documentation	Examples
ps7_globaltimer_0	generic		
ps7_gpio_0	gpiops	Documentation	Examples
ps7_gpv_0	generic		
ps7_intc_dist_0	generic		

CDT Build Console [zynq_fsbl]

```
  69844   11336   72500  153680   25850 zynq_fsbl.elf
'Finished building: zynq_fsbl.elf.size'
' '

19:05:16 Build Finished (took 15s.540ms)
```

Figure 102: SDK after Generating the FSBL

Take a look inside “lscript.ld”:



Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory.
In this page, you can define new memory regions, and change the assignment of sectionsto memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Stack and Heap Sizes

Stack Size	0x6000
Heap Size	0x2000

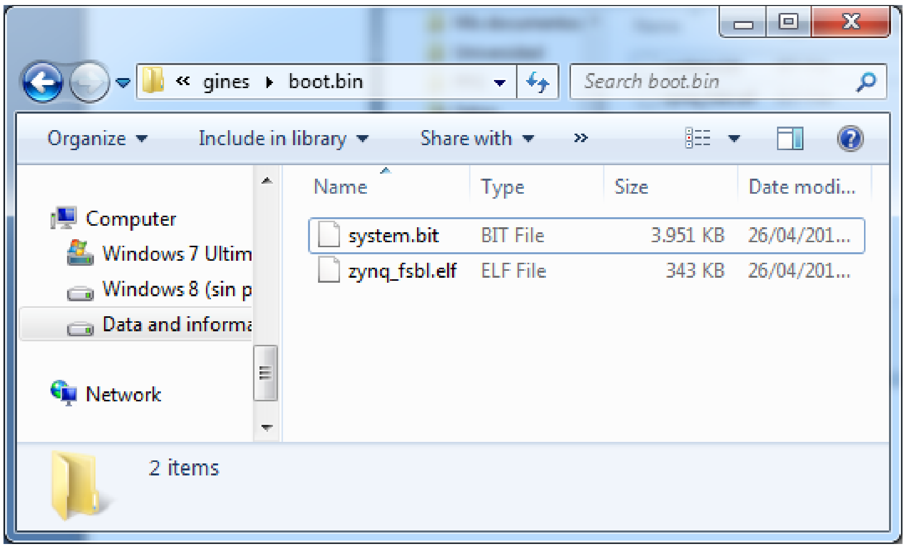
Section to Memory Region Mapping

Section Name	Memory Region

Figure 103: “lscript.ld” File

Notice that in the Hardware Memory Map, two memories are listed. The "ps7_ram_0_S_AXI_BASEADDR" is the ARM local OCM memory location following a reset. This memory runs at the processor speed and is, therefore, one of the fastest types of memory available to the ARM Cortex-A9 cores. The second memory segment is listed as "ps7_ram_1_S_AXI_BASEADDR". This is part of the same ARM local OCM.

Finally, copy the file "zynq_fsbl.elf" which can be found in the path: .\linuxInZedBoard\linuxInZedBoard.sdk\SDK\SDK_Export\zynq_fsbl\Debug\zynq_fsbl.elf and paste it in another folder, for instance, a folder called "boot.bin". Copy also the BitStream file "system.bit" in the same folder:



Name	Type	Size	Date modi...
system.bit	BIT File	3.951 KB	26/04/201...
zynq_fsbl.elf	ELF File	343 KB	26/04/201...

2 items

Figure 104: "boot.bin" Folder after FSBL

2.6.2. Second Stage Boot Loader, U-Boot

As already mentioned, there are three different methods to get the desired file "u-boot.elf" and the three provide exactly the same result and create the same "u-boot.elf" file.

- Download the official Xilinx U-Boot repository, configure it for the ZedBoard and build the required file.
- Download the U-boot BSP generator for the Xilinx git repository, add it to Xilinx SDK and built the required file.
- Configure BuildRoot to automatically download and configure a specific U-Boot repository and to build the file (recommended).

Remember that is only necessary to implement the last method, the rest are optional because of their same result. Moreover, a Linux Operating Systems will be necessary to continue the process. In this case, the CentOS OS will be used in the Virtual Machine VMware Player. How install and configure the program is showed in "Appendix 2: Prerequisites".

2.6.2.1. First Method (optional) – U-Boot Repository

This method is showed in order to better understanding of how U-Boot is built, due to the fact that BuildRoot carries out automatically these steps to build U-Boot (third method).

2.6.2.1.1. VMware Player

Once VMware Player is installed and all the sources are downloaded, and before building the U-Boot file, a cross compiler is required. It is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. In this case, the Sourcery Codebench will be used as cross compiler, which is a complete development environment for embedded C/C++ development on ARM, among others.

In order not to extend more this guide, how to install it and configure it is realized in "Appendix 2: Prerequisites". Consequently, the following items are considered as already done:

1. Linux environment variable "CROSS_COMPILE" has to be set to "arm-xilinx-linuxgnueabi-".
2. The PATH environment variable has to be set to <location_of_the_folder> /CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_GNU_Linux/bin:$PATH.
3. The ARCH environment variable has to be set to arm.

Once the Cross Compiler is configured, open the "terminal" program:

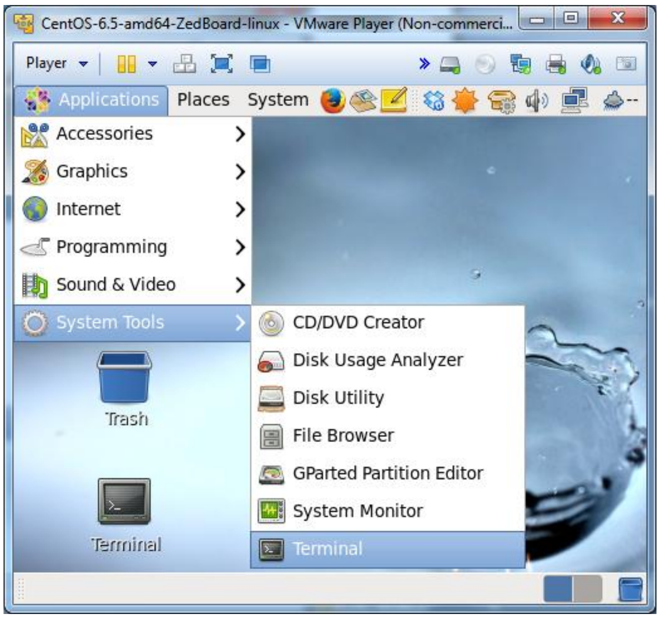


Figure 105: Linux "Terminal" Window in VMware Player

Once the Cross Compiler is configured, U-Boot can be configured for one specific platform with the following commands:

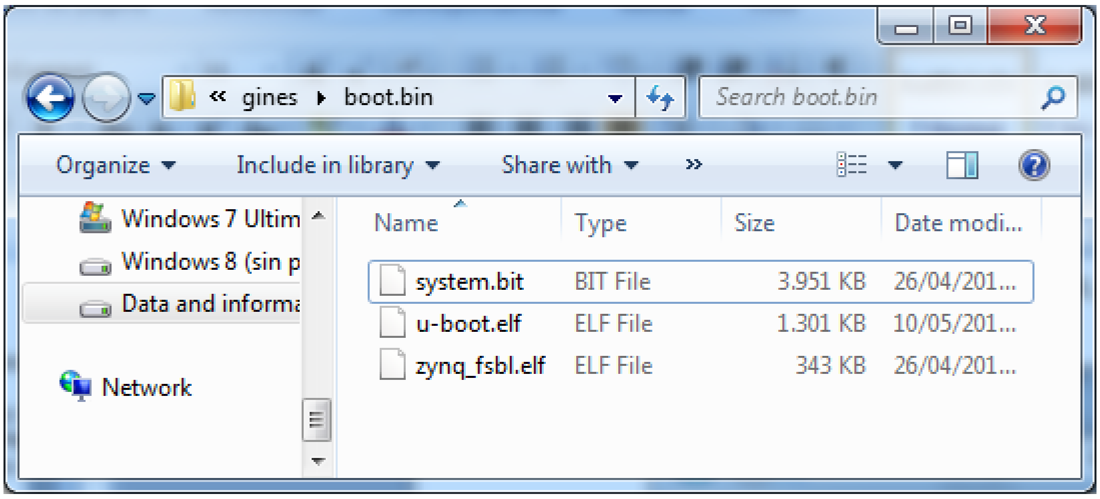
```
cd ~
source .bash_profile
cd ~/u-boot-xlnx/
make distclean
make zynq_zed_config
make
```

Command Window 6: Building U-Boot

A brief description of the function of each command:

- **"cd ~"** simply changes the directory to the main folder.
- **"source .bash_profile"** loads the configuration of the cross compiler.
- **"cd ~/u-boot-xlnx/"** changes the directory to the u-boot-xlnx folder.
- **"make distdean"** cleans any previous configuration in order to prevent errors.
- **"make zynq_zed_config"** configures the U-Boot source tree with the appropriate soft links to select ARM as the target architecture, the ARM v7, the Zynq SoC and the ZedBoard as the target platform.
- **"make"** generates the desired U-Boot file for the Zynq ARM architecture, once the environment variables are properly set for the cross-compiling toolchain.

The Second Stage Boot Loader is created, with the name "u-boot". It should be looked in the "u-boot-diligent" folder, copied, pasted in the windows folder "boot.bin" and renamed as "u-boot.elf".



```
Name            Type       Size       Date modi...
system.bit      BIT File   3.951 KB   26/04/201...
u-boot.elf      ELF File   1.301 KB   10/05/201...
zynq_fsbl.elf   ELF File     343 KB   26/04/201...
```

Figure 106: "boot.bin" Folder after Adding the "u-boot.elf" File

2.6.2.2. Second Method (optional) – U-Boot BSP generator for Xilinx

This method will not be displayed in order to not to expend more this guide, due to it provides the same result as the previous and following ones, and it does not contribute with a new knowledge, due to the idea and the concept is the same as the previous one.

2.6.2.3. Third Method – U-Boot Using BuildRoot

BuildRoot will generate the U-Boot file, the Device Tree Binary file, the Linux Kernel file file and the Root File System Image file. Because of that, all subchapters related to BuildRoot will be merged in the subchapter "3.7. BuildRoot".

2.6.3. Boot Image, boot.bin

The boot image will be created in SDK using the FSBL, the hardware BitStream file and the U-Boot ELF file. This file will be written in the SD Card for allowing to the ZedBoard being booted into U-Boot. Before carrying out this chapter, remember that subchapter "3.7. BuildRoot" must be finished to get the desired U-Boot file.

2.6.3.1. Software Development Kit (SDK)

Whether SDK was closed, it must be re-launched with the previous workspace. Once opened, click in "Xilinx Tools" and "Create Zynq Boot Image":

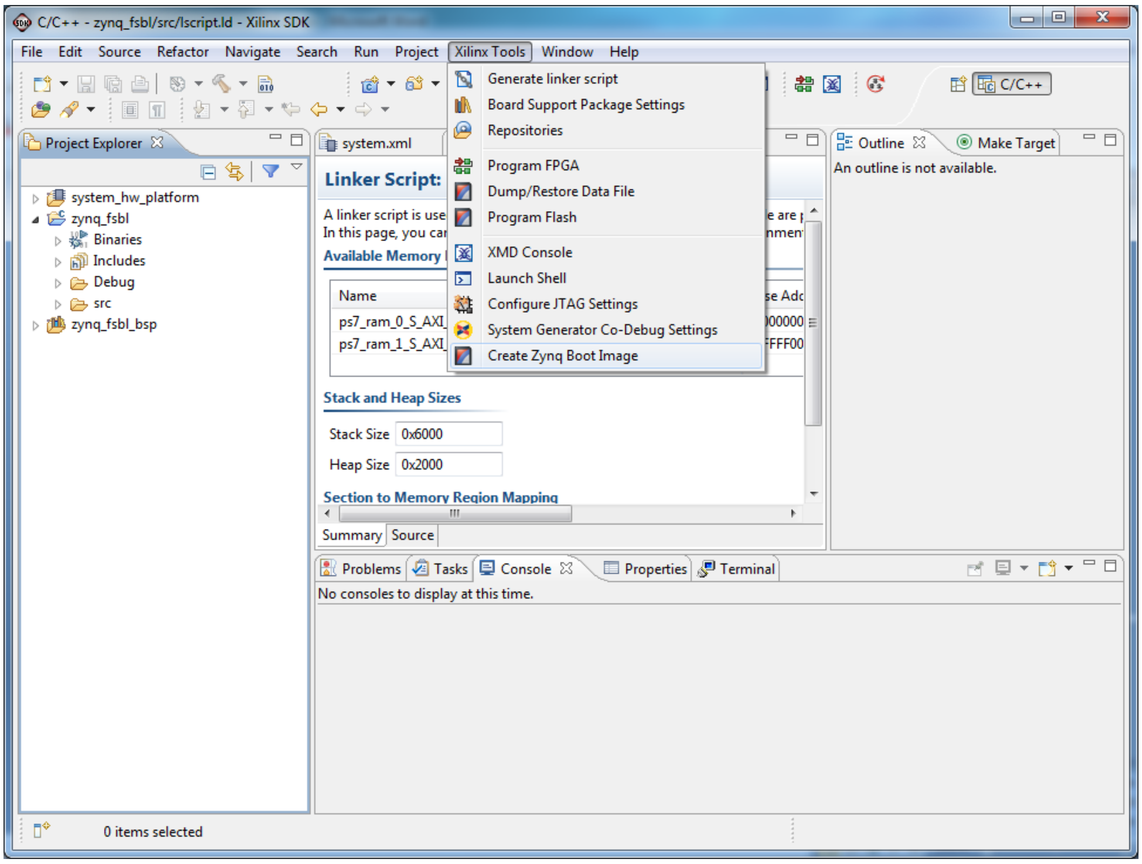
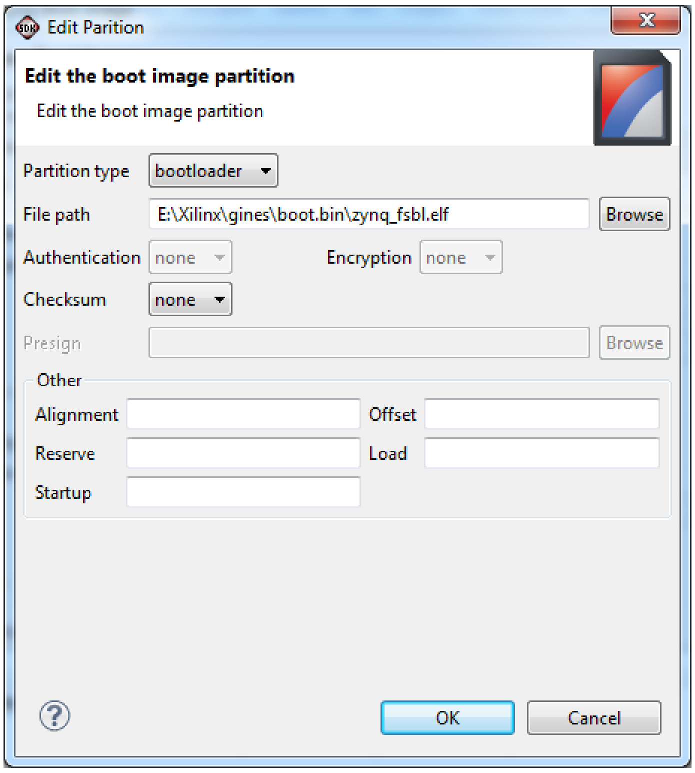


Figure 107: Create Zynq Boot Image Selection

For adding the Boot image partitions, click in "Add" three times, adding each file as showed in the following figures. It is important to remember that the order of the images should always be the same.

1. First Stage Boot Loader ("zynq_fsbl.elf").
2. The Programmable Logic BitStream ("system.bit").
3. The software application file, in this case, the Second Stage Boot Loader U-Boot ("u-boot.elf"). Remember that this file will be obtained in subchapter "3.7. BuildRoot".

Fill in the information such as the next figures, by dicking in "Add":


Edit Parition

Edit the boot image partition

Edit the boot image partition

Partition type: bootloader

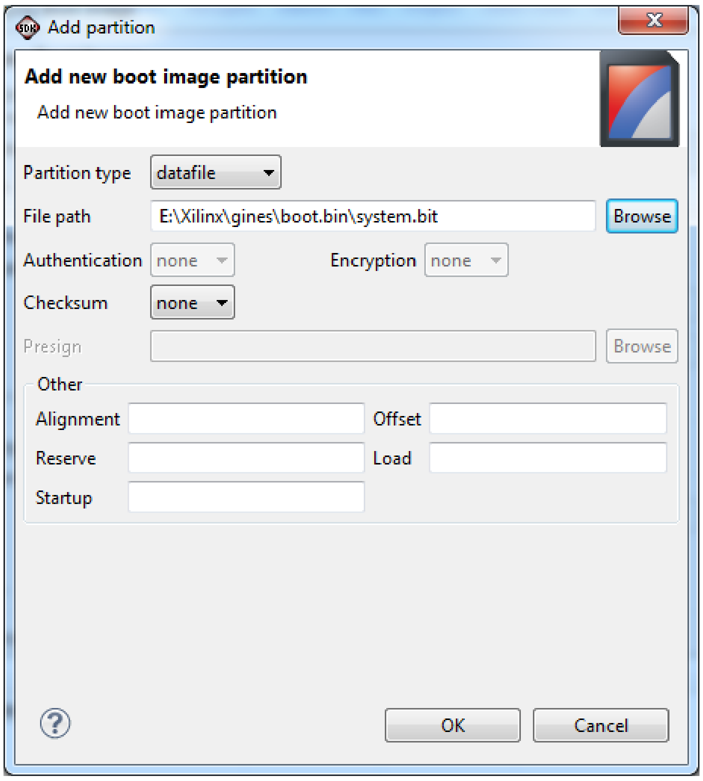
File path: E:\Xilinx\gines\boot.bin\zynq_fsbl.elf

Authentication: none

Encryption: none

Checksum: none

Figure 108: Adding the FSBL


Add partition

Add new boot image partition

Add new boot image partition

Partition type: datafile

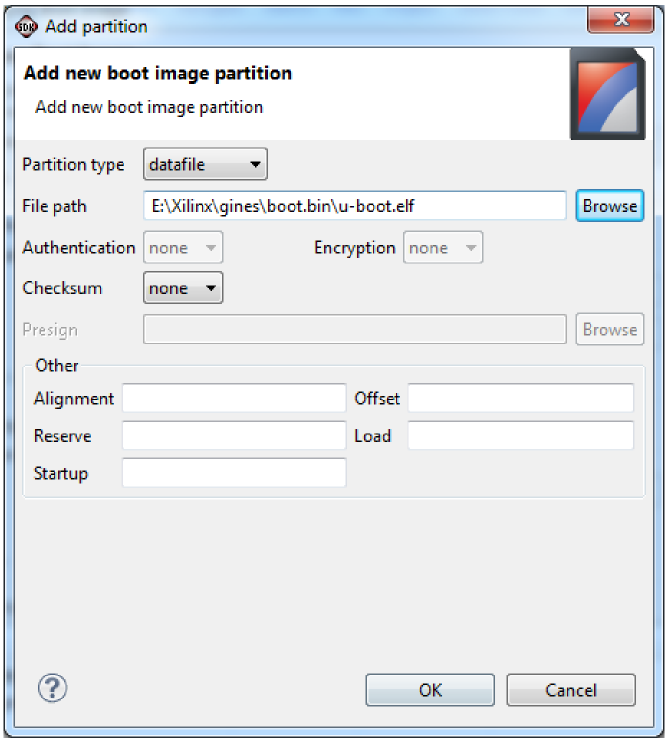
File path: E:\Xilinx\gines\boot.bin\system.bit

Authentication: none

Encryption: none

Checksum: none

Figure 109: Adding the BitStream File


Add partition

Add new boot image partition

Add new boot image partition

Partition type: datafile

File path: E:\Xilinx\gines\boot.bin\u-boot.elf

Authentication: none — Encryption: none

Checksum: none

Presign:

Other

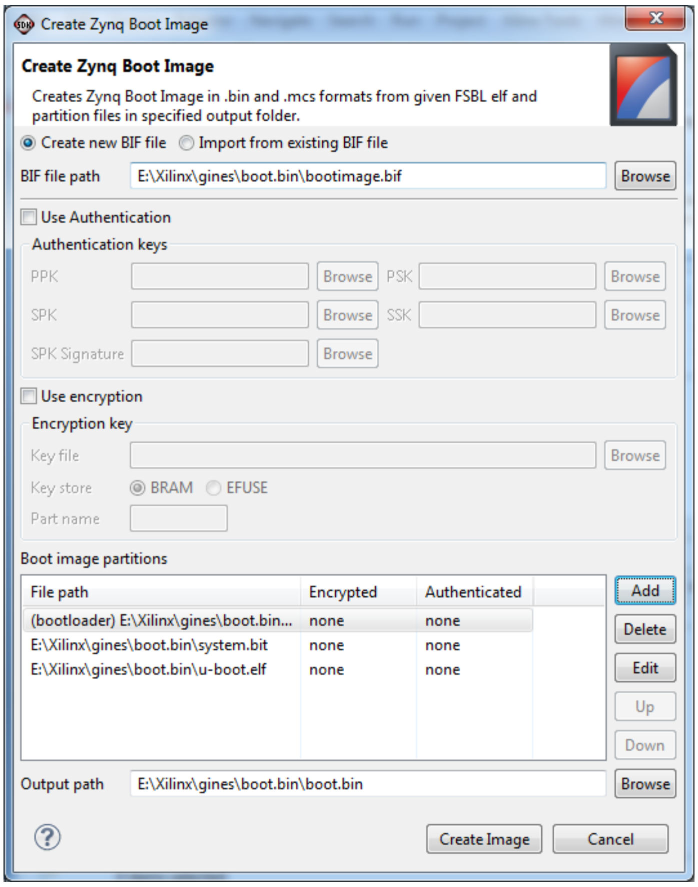
Alignment: — Offset:

Reserve: — Load:

Startup:

Figure 110: Adding the U-Boot File

Finally, the final appearance will be the following:


Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

[✓ Create new BIF file] Import from existing BIF file

BIF file path: E:\Xilinx\gines\boot.bin\bootimage.bif

Use Authentication

Authentication keys

PPK — PSK

SPK — SSK

SPK Signature

Use encryption

Encryption key

Key file

Key store: [✓ BRAM] EFUSE

Part name

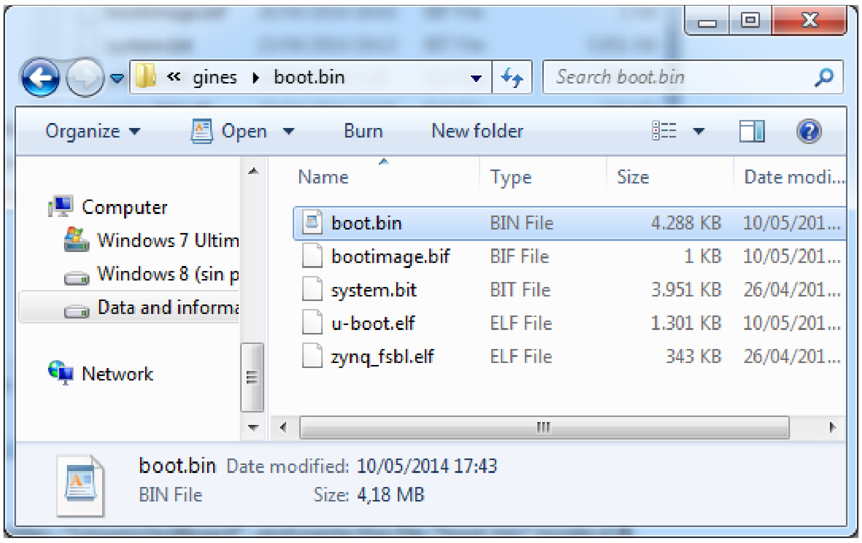
Boot image partitions

File path	Encrypted	Authenticated
(bootloader) E:\Xilinx\gines\boot.bin...	none	none
E:\Xilinx\gines\boot.bin\system.bit	none	none
E:\Xilinx\gines\boot.bin\u-boot.elf	none	none

Output path: E:\Xilinx\gines\boot.bin\boot.bin

Figure 111: Create the Zynq Boot Image

Finally, click in "Create Image". Once SDK has been concluded, the file "boot.bin" will be found in the same folder.

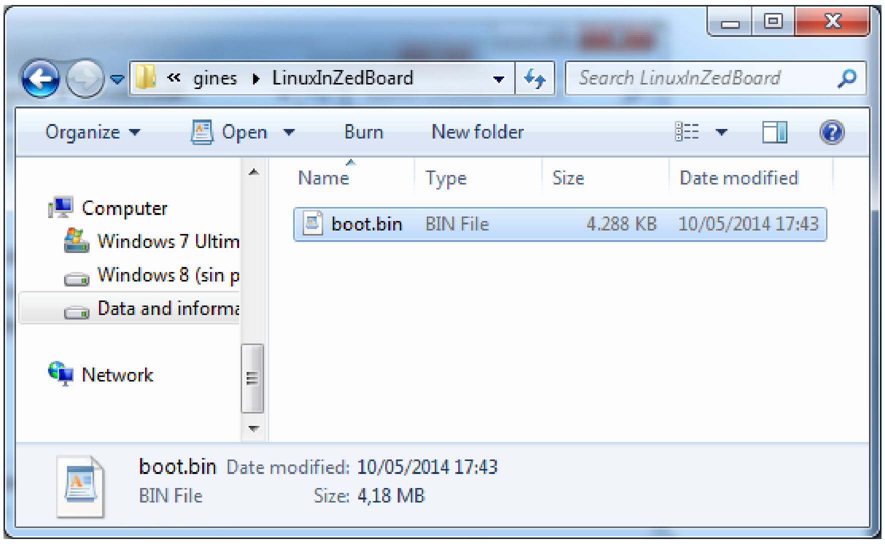

gines ▸ boot.bin

Name	Type	Size	Date modi...
boot.bin	BIN File	4.288 KB	10/05/201...
bootimage.bif	BIF File	1 KB	10/05/201...
system.bit	BIT File	3.951 KB	26/04/201...
u-boot.elf	ELF File	1.301 KB	10/05/201...
zynq_fsbl.elf	ELF File	343 KB	26/04/201...

boot.bin Date modified: 10/05/2014 17:43
BIN File Size: 4,18 MB

Figure 112: "boot.bin" Folder

Create a new folder, "LinuxInZedBoard", and paste the file "boot.bin" inside it:


gines ▸ LinuxInZedBoard

Name	Type	Size	Date modified
boot.bin	BIN File	4.288 KB	10/05/2014 17:43

boot.bin Date modified: 10/05/2014 17:43
BIN File Size: 4,18 MB

Figure 113: "LinuxInZedBoard" Folder

2.6.3.2. Tera Term (optional)

"boot.bin" can also be copied into the SD Card memory and tested in the board. Notice that the SD Card has to be previously formatted in FAT32 file system. For tested the boot file, "Tera Term" can be used. Once Tera Term is launched, switch on the board and click in "File", "New conection…", "Serial" and "OK".

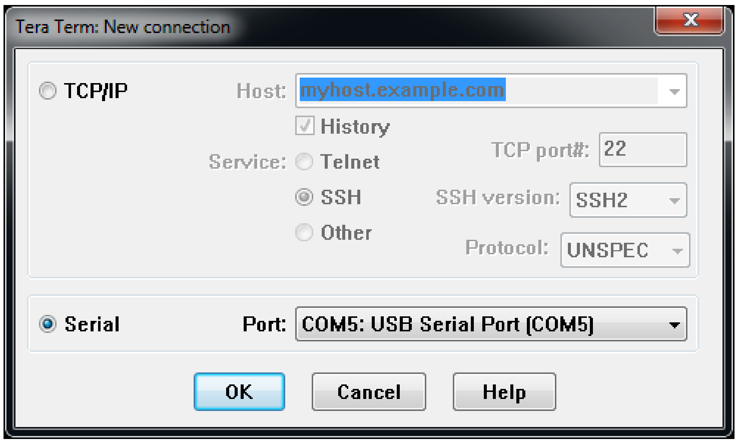
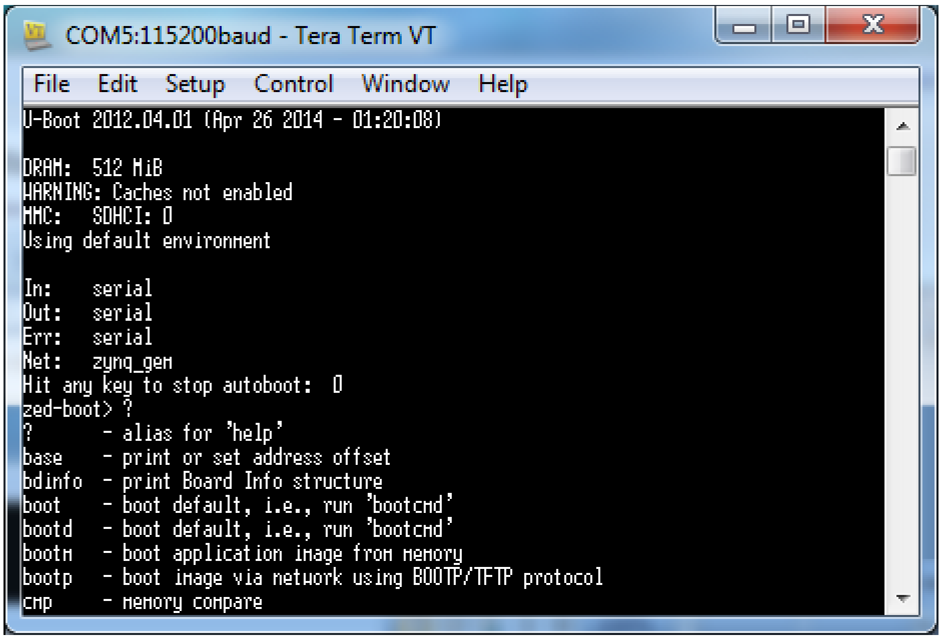


Figure 114: Tera Term - New Connection

Now, click in the reset button of the ZedBoard or switch it off and switch it on again. If no button is pushed, the boot loader will try to load the OS but errors will be displayed because there is no OS inside the SD Card. Nevertheless, whether a button in the keyboard is pushed, the boot loaded will not load the operating system and the user will be able to use the command plot. Writing "?" in the command plot, the command list will be showed. Therefore, the boot file works properly.


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
U-Boot 2012.04.01 (Apr 26 2014 - 01:20:08)

DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   SDHCI: 0
Using default environment

In:    serial
Out:   serial
Err:   serial
Net:   zynq_gem
Hit any key to stop autoboot:  0
zed-boot> ?
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
```

Figure 115: Testing the "boot.bin" File

2.7. Device Tree Binary, "devicetree.dtb"

2.7.1. Device Tree Source (DTS File)

Remember that there are also three methods to get the DTS file.

- Download the Linux Device Tree Generator, add it to Xilinx SDK and build the custom DTS file (recommended for custom hardware configuration).

- Download the official Linux Xilinx repository linux-xlnx, which already contains the default DTS file.
- Configure BuildRoot to automatically compile the DTB file from the default DTS file for ZedBoard (recommended for default hardware configuration and recommended in this case).

2.7.1.1. First Method (optional) – Linux Device Tree Generator

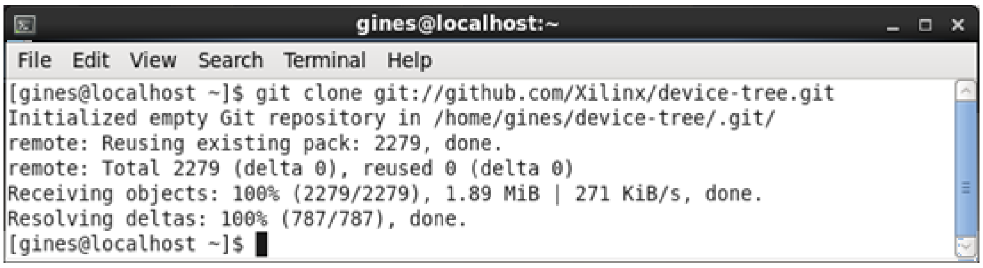
2.7.1.1.1. VMware Player

The Device Tree Generator (dtg) is not included in the tools installed as default. It has to be downloaded from the Xilinx Github Repository. For this purpose, the "Terminal" program must be open in VMware Player and the git repository can be downloaded with the following command:

```
git clone git://github.com/Xilinx/device-tree.git
```

Command Window 7: Downloading the Device Tree Generator

An image of this process is the following:



```
[gines@localhost ~]$ git clone git://github.com/Xilinx/device-tree.git
Initialized empty Git repository in /home/gines/device-tree/.git/
remote: Reusing existing pack: 2279, done.
remote: Total 2279 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (2279/2279), 1.89 MiB | 271 KiB/s, done.
Resolving deltas: 100% (787/787), done.
[gines@localhost ~]$ 
```

Figure 116: Downloading the Device Tree Generator

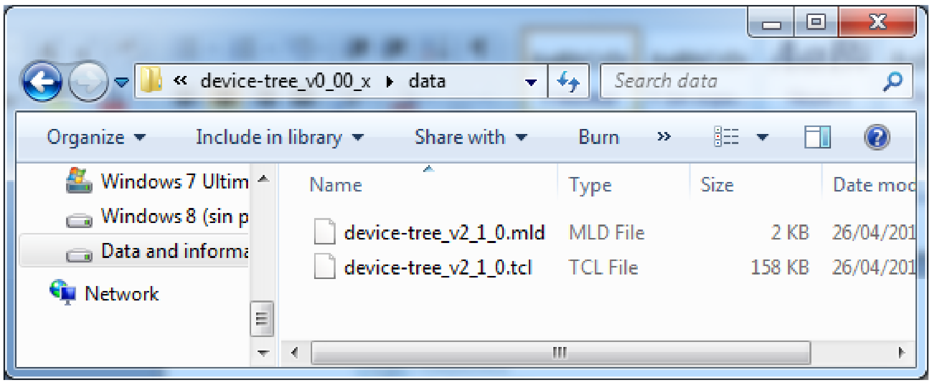
A folder called "device-tree" with two files will be downloaded. The two files are "device-tree_vX.mld" and "device-tree_vX.tcl". Copy the two files from the Linux OP to Windows and paste them in the following directory:

<Xilinx_ise_installation_path>\ISE_DS\EDK\sw\lib\bsp\device-tree_v0_00_x\data\

For instance, a full path can be the next:

E:\Xilinx\14.7\ISE_DS\EDK\sw\lib\bsp\device-tree_v0_00_x\data

The result is showed in the next figure:

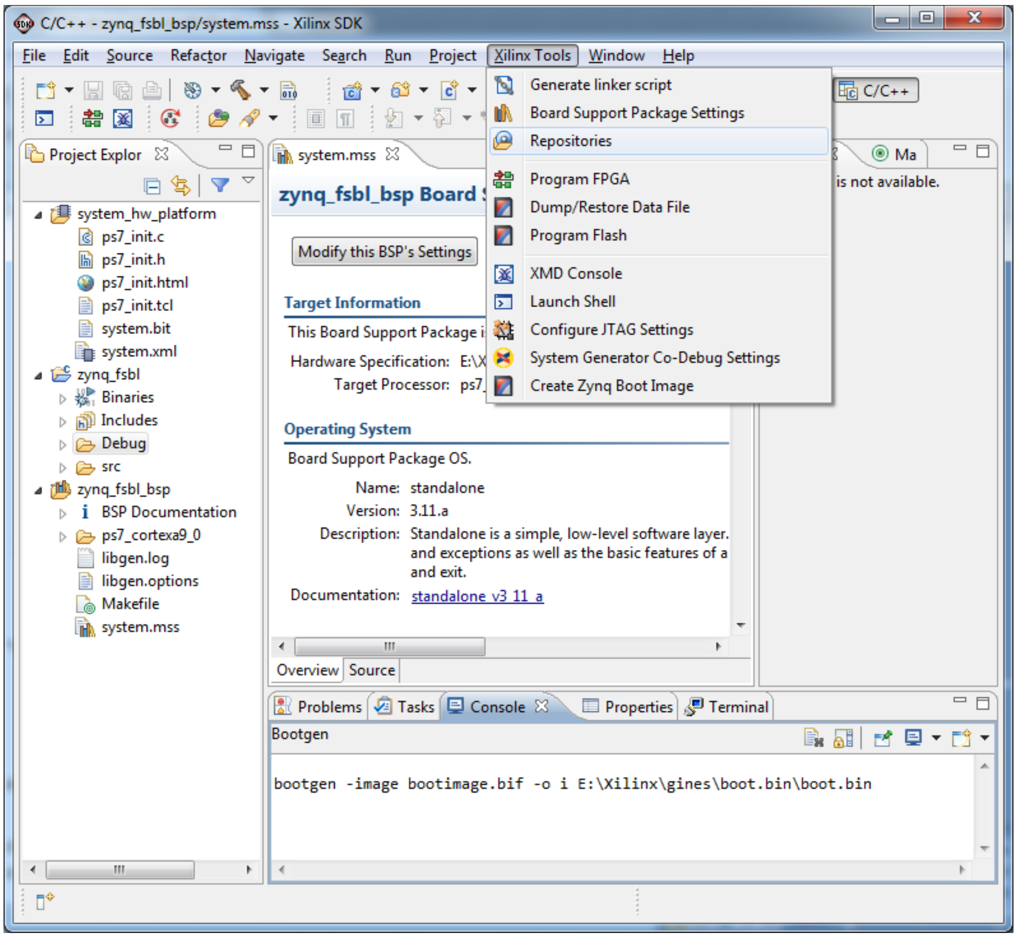


Name	Type	Size	Date mod
device-tree_v2_1_0.mld	MLD File	2 KB	26/04/201
device-tree_v2_1_0.tcl	TCL File	158 KB	26/04/201

Figure 117: Device Tree Generator Folder

2.7.1.1.2. Software Development Kit (SDK)

Now, the BSP repository has to be added in SDK by clicking in "Xilinx Tools" and "Repositories":



Target Information

This Board Support Package i

Hardware Specification: E:\X

Target Processor: ps7_

Operating System

Board Support Package OS.

Name: standalone

Version: 3.11.a

Description: Standalone is a simple, low-level software layer. and exceptions as well as the basic features of a and exit.

Documentation: standalone_v3_11_a

```
bootgen -image bootimage.bif -o i E:\Xilinx\gines\boot.bin\boot.bin
```

Figure 118: Adding the Device Tree Repository 1/2

Click in "New…" in "Global Repositories (available across workspaces)" and select the folder "data" where the two files are inside:

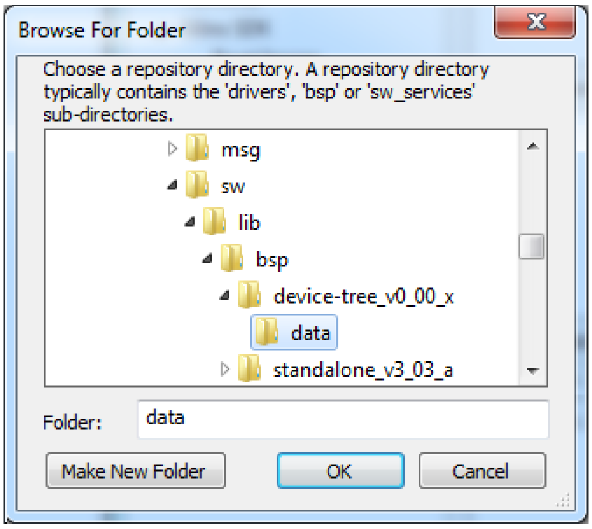
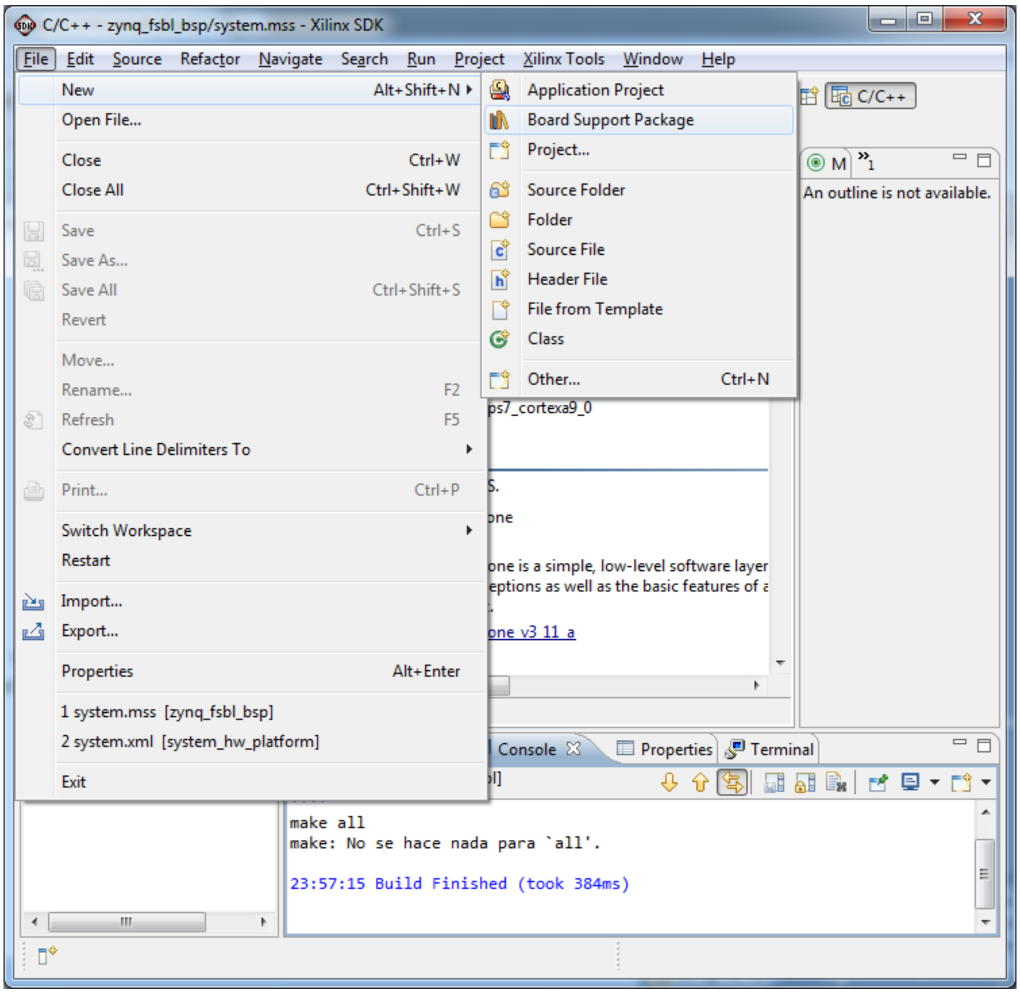

Browse For Folder
Choose a repository directory. A repository directory typically contains the 'drivers', 'bsp' or 'sw_services' sub-directories.
msg
sw
lib
bsp
device-tree_v0_00_x
data
standalone_v3_03_a
Folder: data
Make New Folder
OK
Cancel

Figure 119: Adding the Device Tree Repository 2/2

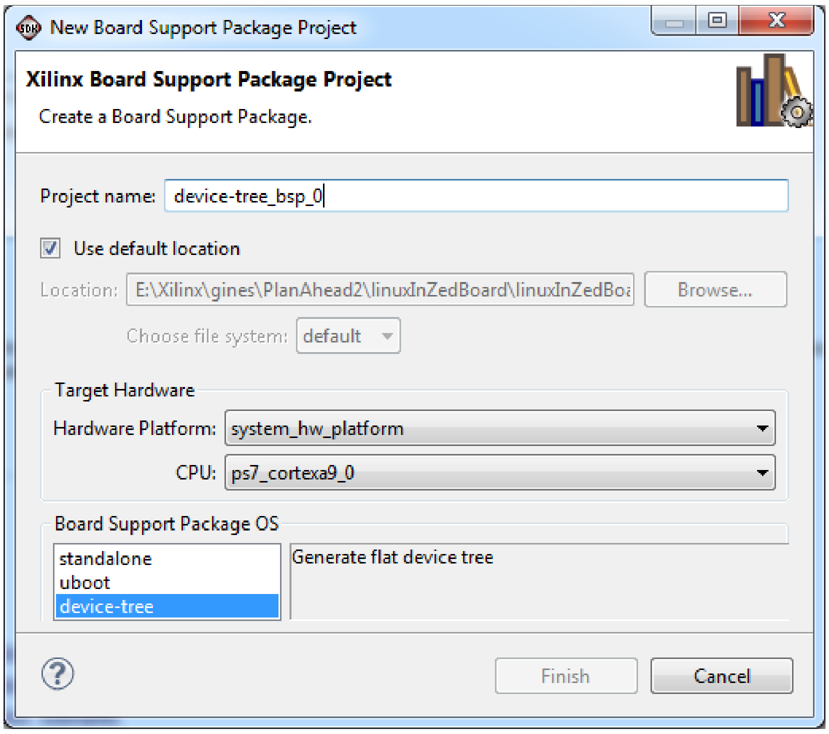
Finally, "OK". When the "Console" finish, "File", "New" and "Board Support Package":



```
make all
make: No se hace nada para `all'.

23:57:15 Build Finished (took 384ms)
```

Figure 120: Creating the DBS File 1/3

Now, a new wizard appears to select what short of BSP (Board Support Package) the user wants. Select "device-tree" and "Finish":


New Board Support Package Project
Xilinx Board Support Package Project
Create a Board Support Package.
Browse...
Finish
Cancel

Project name: device-tree_bsp_0

☑ Use default location

Location: E:\Xilinx\gines\PlanAhead2\linuxInZedBoard\linuxInZedBo

Choose file system: default

Target Hardware

Hardware Platform: system_hw_platform

CPU: ps7_cortexa9_0

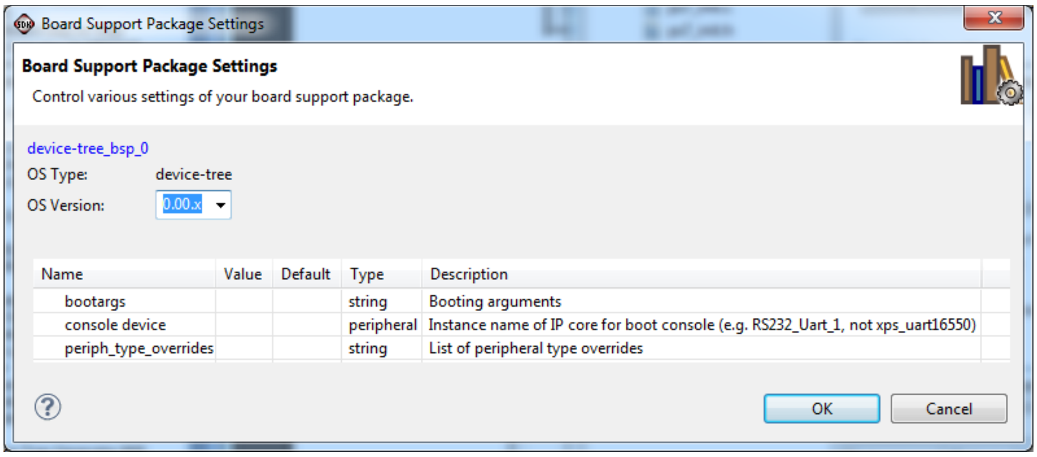
Board Support Package OS

standalone
uboot
device-tree

Generate flat device tree

Figure 121: Creating the DBS File 2/3

A new window appears, for configuring the "bootargs", the "console device" and the "periph_type_overrides". They are going to be left in blank for the moment and "OK". Typically they are left empty in .dts source files and populated at boot time.


Board Support Package Settings
Board Support Package Settings
Control various settings of your board support package.
OK
Cancel

device-tree_bsp_0

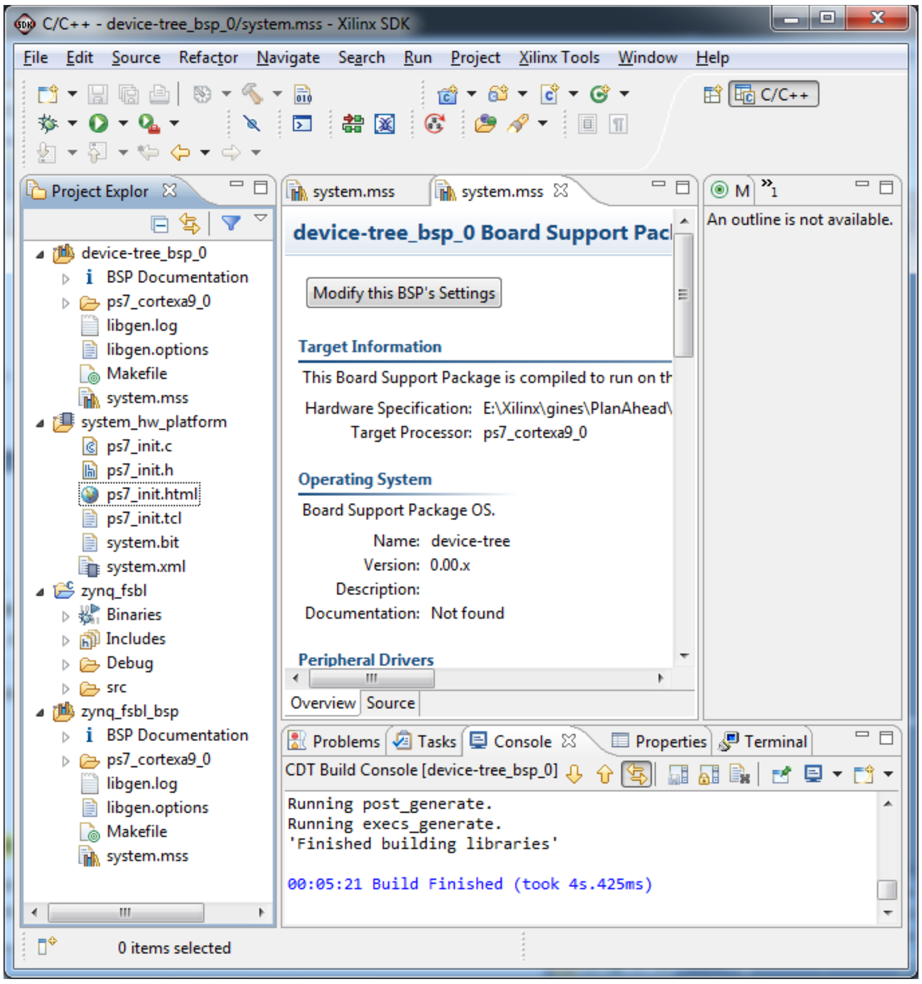
OS Type: device-tree

OS Version: 0.00.x

Name	Value	Default	Type	Description
bootargs			string	Booting arguments
console device			peripheral	Instance name of IP core for boot console (e.g. RS232_Uart_1, not xps_uart16550)
periph_type_overrides			string	List of peripheral type overrides

Figure 122: Creating the DBS File 3/3

The device-tree-BSP has been added to the SDK workspace and displays it in the Project Explorer:



This Board Support Package is compiled to run on th
Hardware Specification: E:\Xilinx\gines\PlanAhead\
Target Processor: ps7_cortexa9_0

Board Support Package OS.
Name: device-tree
Version: 0.00.x
Description:
Documentation: Not found

```
CDT Build Console [device-tree_bsp_0]
Running post_generate.
Running execs_generate.
'Finished building libraries'

00:05:21 Build Finished (took 4s.425ms)
```

Figure 123: SDK after Creating the DBS File

After SDK finish rebuilding the project, the new "xilinx.dts" file is created and located in <workspace>/<device-tree-bsp>/<processor-name>/libsrc/devicetree_v0_00_x folder. This file is the Device Tree Source (dts):

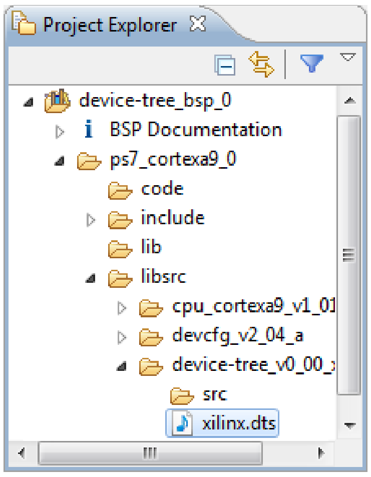
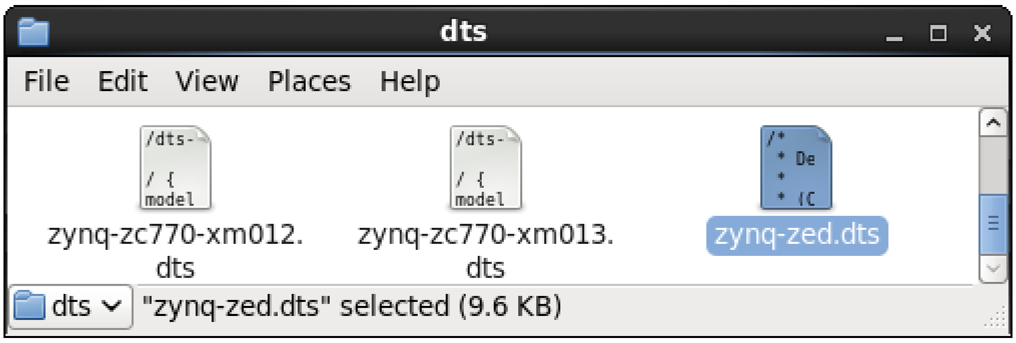


Figure 124: File Location of the DTS File

2.7.1.2. Second Method (optional) – Linux Xilinx Repository “xilinx-xlnx”

The second method, downloading the linux-xlnx repository (see “Appendix 2: Prerequisites”), directly provides the DTS file of almost available boards/architectures. All these TDS files can be found in:

.../linux-xlnx/arch/arm/boot/dts/zynq-zed.dts



"zynq-zed.dts" selected (9.6 KB)

Figure 125: Linux-xlnx Folder with the Default DTS Files

2.7.1.3. Third Method – DTS File Using BuildRoot

BuildRoot includes the default DTS file, because it automatically downloads the desired Git Repository, in this case the Linux-xlnx repository, which includes the default DTS file for the ZedBoard. Therefore, no extra steps are required.

2.7.2. Device Tree Binary (DTB File)

Remember that it is necessary to convert the human-readable file DTS in a proper binary machine-readable file DTB. One again, there are 3 different options.

- Download the Device Tree Compiler (dtc) for Linux and build the DTB file.
- Use the dtc available in the Linux-xlxn folder and create it manually like in the previous method.
- Use BuildRoot, when the Kernel Image and/or the Root File System Image are built, the DTB file is automatically created from the pre-indicated DTS source (recommended).

The linux-xlxn folder has its own dtc compiler; therefore, it is completely unnecessary to use the first method because the second method is configured and executed in the same way, since the dtc used in each one is the same. In addition, the third method realizes the same steps than the second method, but it executes them automatically. Therefore, the second method will be shown in order to better understanding the third method.

2.7.2.1. Second Method (optional) – DTB File from the Linux-xlnx dtc

2.7.2.1.1. VMware Player

The device tree compiler (dtc), located under scripts/dtc in the Linux kernel source, is responsible for carrying out this work. It can compile the DTS file into a DTB file with the following command:

```
      ~/linux-xlnx/scripts/dtc/dtc -I dts -O dtb -o
<output_file_path>/devicetree.dtb <input_file_path>/xilinx.dts
```

Command Window 8: Building the DTB File from the DTS File 1/2

There is another method with the dtc compiler which returns the same DTB file. For this purpose, the “xilinx.dts” file must be copied into the next Linux path:

.../linux-xlnx/arch/arm/boot/dts/xilinx.dts

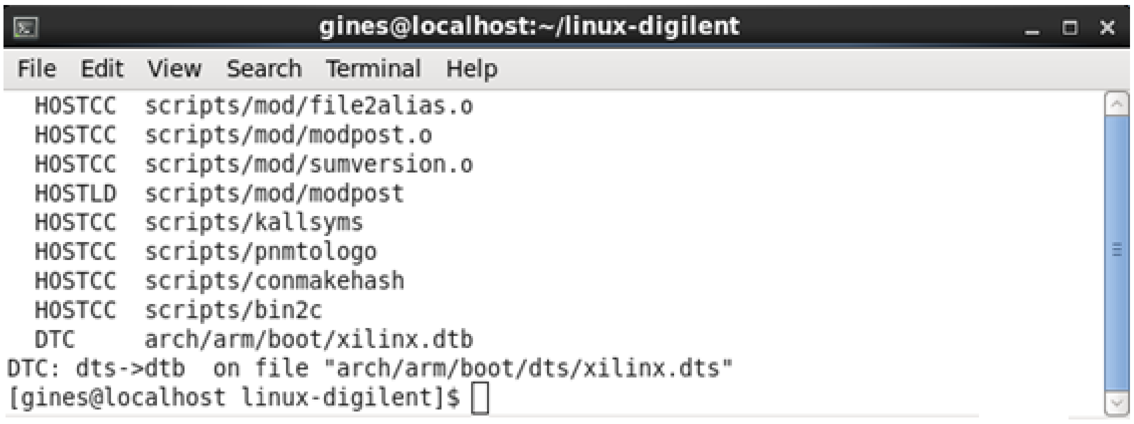
Whether this folder is opened, a large amount of DTS files will be looked. For compiling a DTS file into a DTB the commands are the following.

```
      cd ~/linux-xlnx/

      make ARCH=arm distclean

      make ARCH=arm CROSS_COMPILE=arm-linux- xilinx_zynq_defconfig

      make ARCH=arm xilinx.dtb
```

Command Window 9: Building the DTB File from the DTS File 2/2

- **“cd ~/linux-xlnx/”** selects the linux-xlnx directory.
- **“make ARCH=arm distclean”** runs a make distribution clean command against the kernel source code for good measure. This command will remove all intermediary files created by setting as well as any intermediary files created by make and it is a good way to clean up any stale configurations.
- **“make ARCH=arm CROSS_COMPILE=arm-linux- xilinx_zynq_defconfig”** configures the Linux-xlnx folder for the ZedBoard, like U-Boot and the Linux kernel. The command prepares the kernel source tree for the Zynq-7000 architecture including some special configuration for the ZedBoard. It builds a default configuration “.config” file. The architecture type and the cross compiler prefix are also specified respectively with “ARCH=arm” and “CROSS_COMPILER=arm-linux-”.
- **“make ARCH=arm xilinx.dtb”** order to the dtc compiles to convert the DTS file into a DTB file.

The result in the "Terminal" windows is the next:


gines@localhost:~/linux-digilent
File Edit View Search Terminal Help

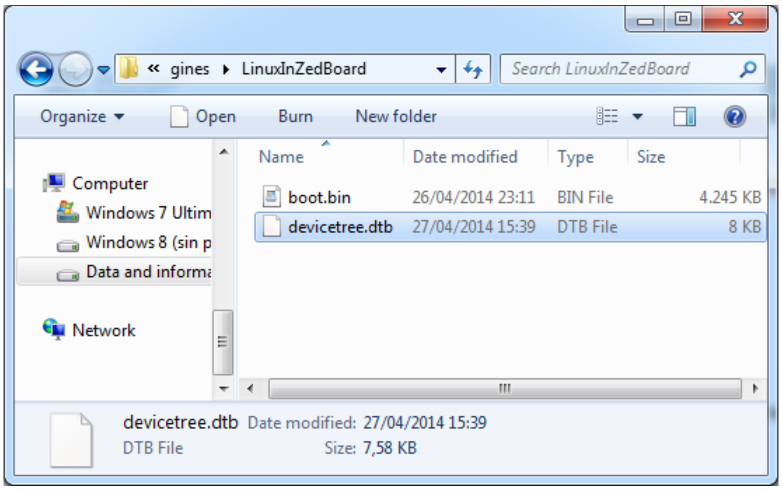
```
  HOSTCC  scripts/mod/file2alias.o
  HOSTCC  scripts/mod/modpost.o
  HOSTCC  scripts/mod/sumversion.o
  HOSTLD  scripts/mod/modpost
  HOSTCC  scripts/kallsyms
  HOSTCC  scripts/pnmtologo
  HOSTCC  scripts/conmakehash
  HOSTCC  scripts/bin2c
  DTC     arch/arm/boot/xilinx.dtb
DTC: dts->dtb  on file "arch/arm/boot/dts/xilinx.dts"
[gines@localhost linux-digilent]$ 
```

Figure 126: DTB File Generated

This command will search the "xilinx.dts" file in the previous indicated path, linux-xlnx/arch/arm/boot/dts/, and it will generate the "xilinx.dtb" file in the next folder:

.../linux-xlnx/arch/arm/boot/

Once the DTB file is generated with any of these two ways, copy it, paste into the Windows folder "LinuxInZedBoard" and rename it to "devicetree.dtb".



Name	Date modified	Type	Size
boot.bin	26/04/2014 23:11	BIN File	4.245 KB
devicetree.dtb	27/04/2014 15:39	DTB File	8 KB

devicetree.dtb Date modified: 27/04/2014 15:39
DTB File Size: 7,58 KB

Figure 127: "LinuxInZedBoard" Folder

2.8. Linux Kernel File, "uImage"

Once again, there are **different methods** to build the Linux Kernel file:

- Using the linux-xlnx directory.
- Using BuildRoot to configure and build it.

It will be showed how the linux-xlnx directory builds the Kernel file in order to better understanding how BuildRoot performs this task automatically.

2.8.1. First Method (optional) – Linux Kernel Using the Linux-xlnx Directory

2.8.1.1. VMware

Using the **first method**, the required commands are showed below:

```
cd ~
source .bash_profile
cd ~/linux-xlnx/
make distclean
make ARCH=arm CROSS_COMPILE=arm-linux- xilinx_zynq_defconfig
make xconfig
make
```

Command Window 10: Building the DTB File 2/2

- **"cd ~"** simply changes the directory to the main folder.
- **"source .bash_profile"** loads the configuration of the cross compiler.
- **"cd ~/linux-xlnx/"** selects the linux-xlnx directory.
- **"make distclean"** runs a make distribution clean command against the kernel source code for good measure. This command will remove all intermediary files created by setting as well as any intermediary files created by make and it is a good way to clean up any stale configurations.
- **"make ARCH=arm CROSS_COMPILE=arm-linux- xilinx_zynq_defconfig"** configures the Linux-xlnx folder for the ZedBoard, like U-Boot and the Linux kernel. The command prepares the kernel source tree for the Zynq-7000 architecture including some special configuration for the ZedBoard. It builds a default configuration ".config" file. The architecture type and the cross compiler prefix are also specified respectively with "ARCH=arm" and "CROSS_COMPILER=arm-linux-".
- **"make xconfig"** allows to the user to customize the Kernel if some special configuration has to be made. Note that it can be also used: "make config", "make menuconfig" or "make gconfig".
- **"make"** generates the desired Linux Kernel file for the Zynq ARM architecture, once the environment variables are properly set.

Once the Kernel has been built, it will be showed in the following path directory.

.../linux-xilinx/arch/arm/boot/uImage

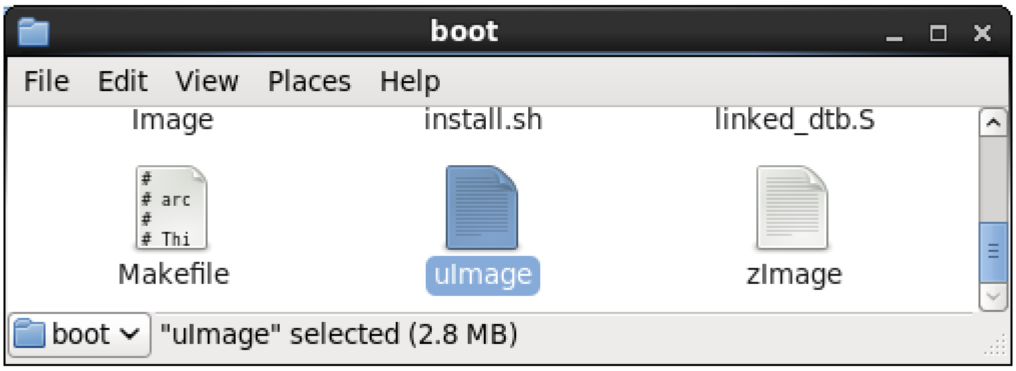


Figure 128: "uImage" Location Folder

2.8.2. Second Method – Linux Kernel Using BuildRoot

2.8.2.1. VMware

Remember that BuildRoot will generate the U-Boot file, the Device Tree Binary file, the Linux Kernel file and the Root File System Image file. Because of that, all subchapters related to BuildRoot will be merged in the subchapter "3.7. BuildRoot".

2.9. Root File System Image

Once again, there are **different methods** to build the Root File System Image:

- Using the linux-xlnx directory.
- Using BuildRoot to configure and build it.

If the File System is build using the linux-xlnx directory, it will require a large amount of time. Nevertheless, BuildRoot automates the process as already mentioned. Therefore, only the second method will be showed.

As happened with U-Boot, the Device Tree Binary and the Kernel, this second method will be explained in the next subchapter.

2.10. BuildRoot

BuildRoot will generate the U-Boot file, the Device Tree Binary file, the Linux Kernel file file and the Root File System Image file. How set BuildRoot in order to configure and customize the Embedded Linux OS will be showed and briefly explained if necessary in this subchapter.

2.10.1. VMware

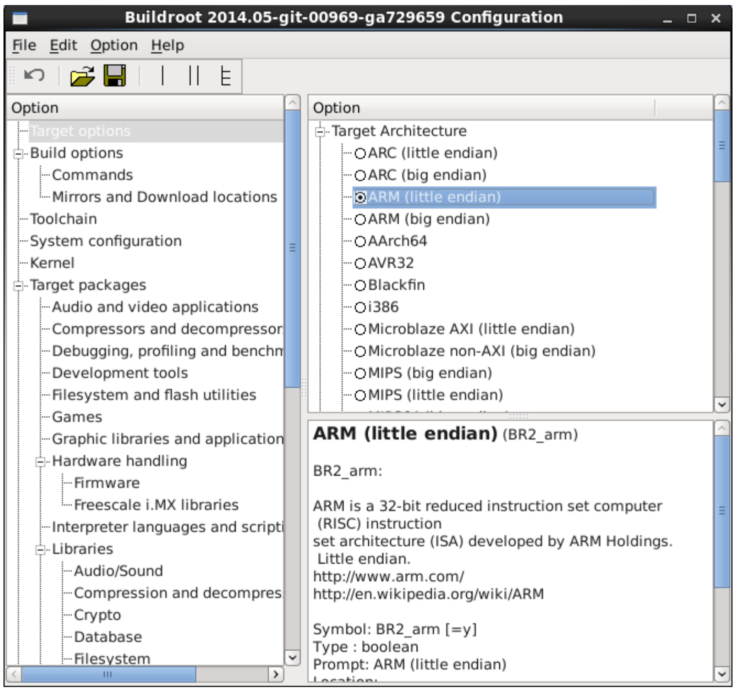
The first step is to pre-configure BuildRoot for the ZedBoard. It can be done with the following commands:

```
cd ~/buildroot/

make distclean

make zedboard_defconfig

make xconfig
```

Command Window 11: Setting BuildRoot for ZedBoard

- **“cd ~/buildroot/”** selects the BuildRoot directory.
- **“make distclean”** runs a make distribution clean command against the kernel source code for good measure. This command will remove all intermediary files created by setting as well as any intermediary files created by make and it is a good way to clean up any stale configurations.
- **“make zedboard_defconfig”** configures the BuildRoot folder for the ZedBoard. The command prepares the kernel source tree for the Zynq-7000 architecture including some special configuration for the ZedBoard. It builds a default configuration “.config” file.
- **“make xconfig”** allows to the user to customize the Kernel if some special configuration has to be made. Note that it can be also used: “make config”, “make menuconfig” or “make gconfig”.

After typing the “make xconfig” command, the configuration must be looked as showed in the next pages. First, the Target Architecture is selected for the ZedBoard: ARM Cortex-A9.



```
Option
  Target options
  Build options
    Commands
    Mirrors and Download locations
  Toolchain
  System configuration
  Kernel
  Target packages
    Audio and video applications
    Compressors and decompressor
    Debugging, profiling and benchm
    Development tools
    Filesystem and flash utilities
    Games
    Graphic libraries and application
    Hardware handling
      Firmware
      Freescale i.MX libraries
    Interpreter languages and scripti
    Libraries
      Audio/Sound
      Compression and decompres
      Crypto
      Database
      Filesystem
```

```
Option
  Target Architecture
    ○ARC (little endian)
    ○ARC (big endian)
    ⦿ARM (little endian)
    ○ARM (big endian)
    ○AArch64
    ○AVR32
    ○Blackfin
    ○i386
    ○Microblaze AXI (little endian)
    ○Microblaze non-AXI (big endian)
    ○MIPS (big endian)
    ○MIPS (little endian)
```

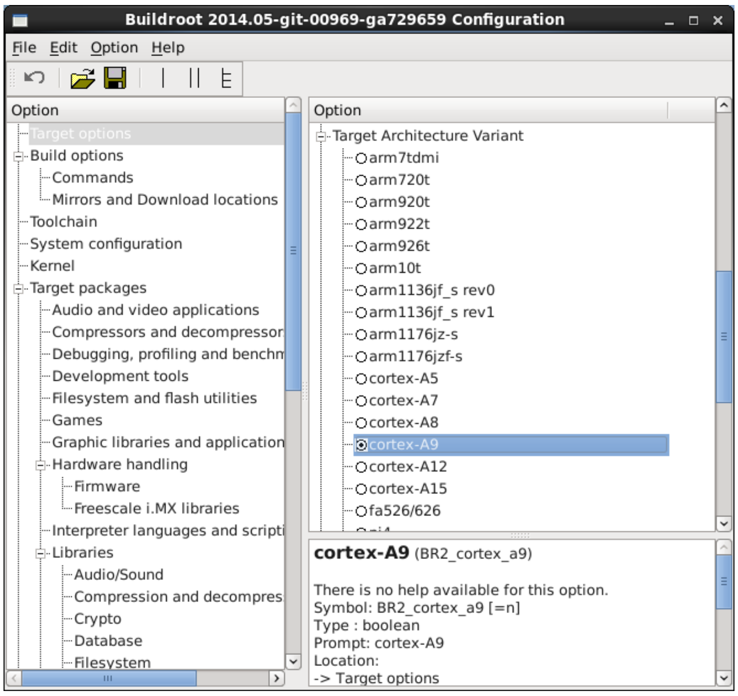
ARM (little endian) (BR2_arm)

BR2_arm:

ARM is a 32-bit reduced instruction set computer
 (RISC) instruction
set architecture (ISA) developed by ARM Holdings.
 Little endian.
http://www.arm.com/
http://en.wikipedia.org/wiki/ARM

Symbol: BR2_arm [=y]
Type : boolean
Prompt: ARM (little endian)

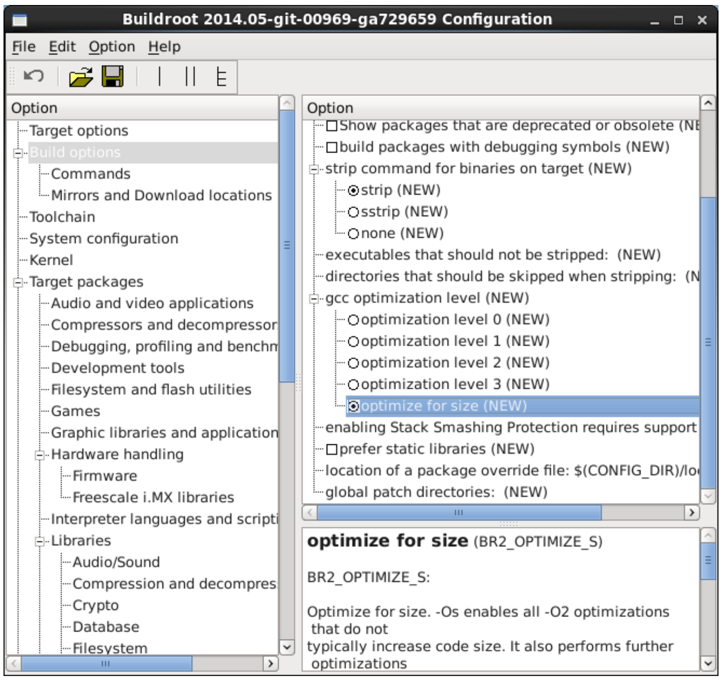
Figure 129: Target Architecture Selection



```
Buildroot 2014.05-git-00969-ga729659 Configuration
File  Edit  Option  Help

Option                                   Option
 Target options                          Target Architecture Variant
 Build options                              ○arm7tdmi
   Commands                                 ○arm720t
   Mirrors and Download locations           ○arm920t
 Toolchain                                  ○arm922t
 System configuration                       ○arm926t
 Kernel                                     ○arm10t
 Target packages                            ○arm1136jf_s rev0
   Audio and video applications             ○arm1136jf_s rev1
   Compressors and decompressor             ○arm1176jz-s
   Debugging, profiling and benchm          ○arm1176jzf-s
   Development tools                        ○cortex-A5
   Filesystem and flash utilities           ○cortex-A7
   Games                                    ○cortex-A8
   Graphic libraries and application        ●cortex-A9
   Hardware handling                        ○cortex-A12
     Firmware                               ○cortex-A15
     Freescale i.MX libraries               ○fa526/626
   Interpreter languages and scripti
   Libraries                             cortex-A9 (BR2_cortex_a9)
     Audio/Sound
     Compression and decompres           There is no help available for this option.
     Crypto                              Symbol: BR2_cortex_a9 [=n]
     Database                            Type : boolean
     Filesystem                          Prompt: cortex-A9
                                         Location:
                                         -> Target options
```

Figure 130: Target Architecture Variant Selection

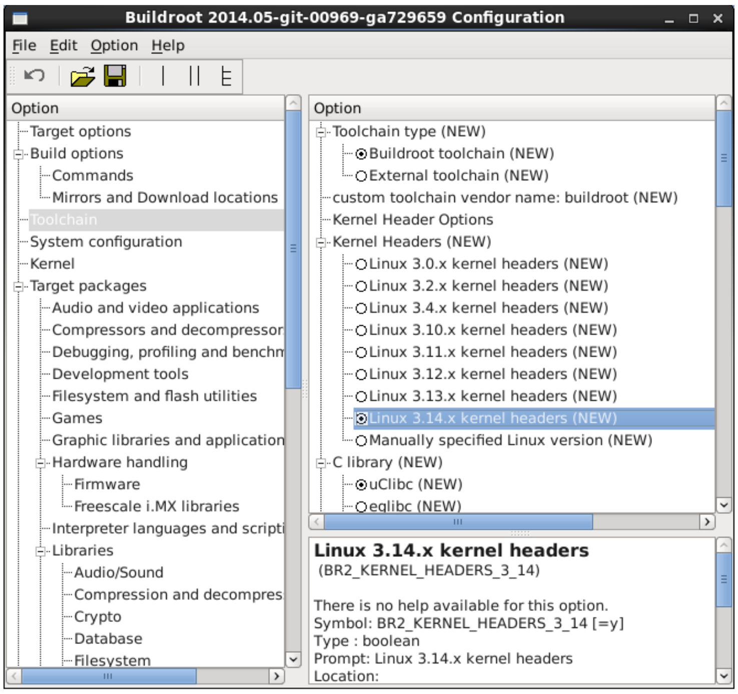
Then, a size optimization should be selected, due to the importance of the memory in embedded systems.



```
Buildroot 2014.05-git-00969-ga729659 Configuration
File  Edit  Option  Help

Option                                   Option
 Target options                          ☐Show packages that are deprecated or obsolete (NE
 Build options                           ☐build packages with debugging symbols (NEW)
   Commands                              strip command for binaries on target (NEW)
   Mirrors and Download locations           ●strip (NEW)
 Toolchain                                  ○sstrip (NEW)
 System configuration                       ○none (NEW)
 Kernel                                  executables that should not be stripped:  (NEW)
 Target packages                         directories that should be skipped when stripping:  (N
   Audio and video applications          gcc optimization level (NEW)
   Compressors and decompressor             ○optimization level 0 (NEW)
   Debugging, profiling and benchm          ○optimization level 1 (NEW)
   Development tools                        ○optimization level 2 (NEW)
   Filesystem and flash utilities           ○optimization level 3 (NEW)
   Games                                    ●optimize for size (NEW)
   Graphic libraries and application     enabling Stack Smashing Protection requires support
   Hardware handling                     ☐prefer static libraries (NEW)
     Firmware                            location of a package override file: $(CONFIG_DIR)/lo
     Freescale i.MX libraries            global patch directories:  (NEW)
   Interpreter languages and scripti
   Libraries                             optimize for size (BR2_OPTIMIZE_S)
     Audio/Sound
     Compression and decompres           BR2_OPTIMIZE_S:
     Crypto
     Database                            Optimize for size. -Os enables all -O2 optimizations
     Filesystem                           that do not
                                         typically increase code size. It also performs further
                                          optimizations
```

Figure 131: Optimization for Size

After that, the cross compiler tool-chain selected is the BuildRoot internal tool-chain which is limited to the usage of the µClibc C library. Linux 3.14.x kernel headers are selected according to the Linux source selected. Moreover, it has compatibility for earlier kernel versions.



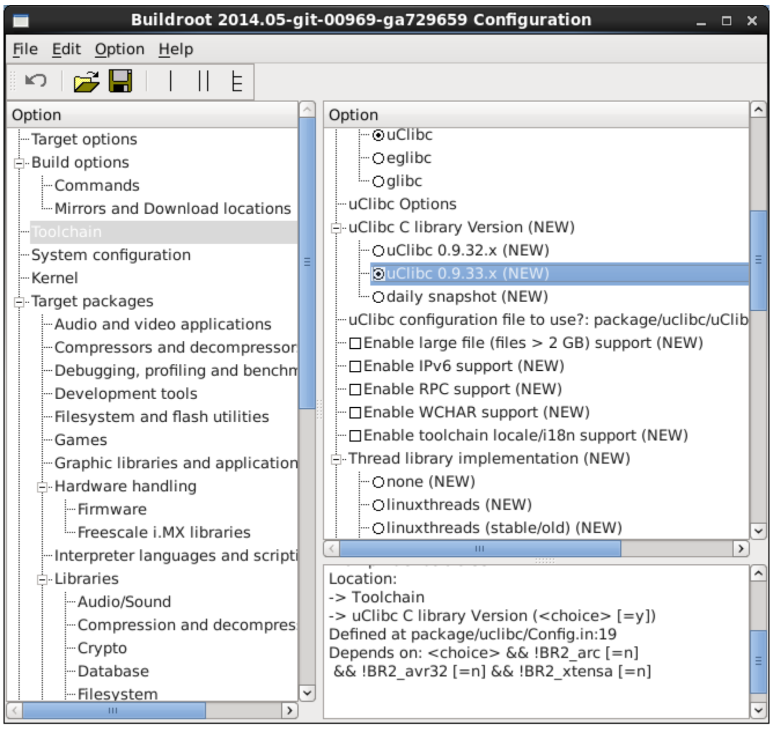
```
Option
 Target options
 Build options
   Commands
   Mirrors and Download locations
 Toolchain
 System configuration
 Kernel
 Target packages
   Audio and video applications
   Compressors and decompressor
   Debugging, profiling and benchm
   Development tools
   Filesystem and flash utilities
   Games
   Graphic libraries and application
   Hardware handling
     Firmware
     Freescale i.MX libraries
   Interpreter languages and scripti
   Libraries
     Audio/Sound
     Compression and decompres
     Crypto
     Database
     Filesystem
```

```
Option
 Toolchain type (NEW)
   ● Buildroot toolchain (NEW)
   ○ External toolchain (NEW)
 custom toolchain vendor name: buildroot (NEW)
 Kernel Header Options
 Kernel Headers (NEW)
   ○ Linux 3.0.x kernel headers (NEW)
   ○ Linux 3.2.x kernel headers (NEW)
   ○ Linux 3.4.x kernel headers (NEW)
   ○ Linux 3.10.x kernel headers (NEW)
   ○ Linux 3.11.x kernel headers (NEW)
   ○ Linux 3.12.x kernel headers (NEW)
   ○ Linux 3.13.x kernel headers (NEW)
   ● Linux 3.14.x kernel headers (NEW)
   ○ Manually specified Linux version (NEW)
 C library (NEW)
   ● uClibc (NEW)
   ○ eglibc (NEW)
```

```
Linux 3.14.x kernel headers
 (BR2_KERNEL_HEADERS_3_14)

There is no help available for this option.
Symbol: BR2_KERNEL_HEADERS_3_14 [=y]
Type : boolean
Prompt: Linux 3.14.x kernel headers
Location:
```

Figure 132: Toolchain and Kernel Headers Selection

Later, µClibc is chosen as C library, due to its BuildRoot Toolchain compatibility.



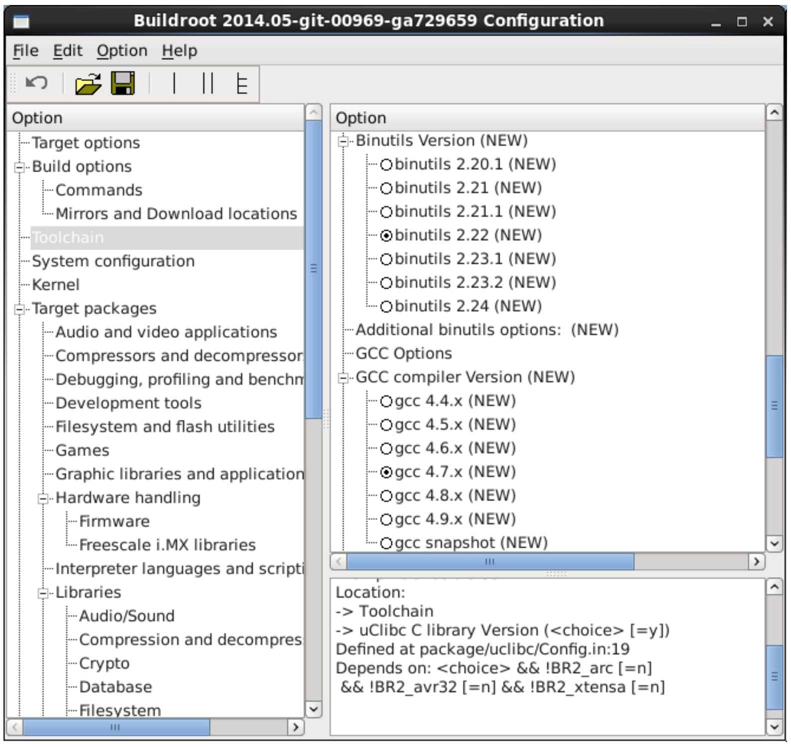
```
Option
 Target options
 Build options
   Commands
   Mirrors and Download locations
 Toolchain
 System configuration
 Kernel
 Target packages
   Audio and video applications
   Compressors and decompressor
   Debugging, profiling and benchm
   Development tools
   Filesystem and flash utilities
   Games
   Graphic libraries and application
   Hardware handling
     Firmware
     Freescale i.MX libraries
   Interpreter languages and scripti
   Libraries
     Audio/Sound
     Compression and decompres
     Crypto
     Database
     Filesystem
```

```
Option
   ● uClibc
   ○ eglibc
   ○ glibc
 uClibc Options
 uClibc C library Version (NEW)
   ○ uClibc 0.9.32.x (NEW)
   ● uClibc 0.9.33.x (NEW)
   ○ daily snapshot (NEW)
 uClibc configuration file to use?: package/uclibc/uClib
 ☐ Enable large file (files > 2 GB) support (NEW)
 ☐ Enable IPv6 support (NEW)
 ☐ Enable RPC support (NEW)
 ☐ Enable WCHAR support (NEW)
 ☐ Enable toolchain locale/i18n support (NEW)
 Thread library implementation (NEW)
   ○ none (NEW)
   ○ linuxthreads (NEW)
   ○ linuxthreads (stable/old) (NEW)
```

```
Location:
-> Toolchain
-> uClibc C library Version (<choice> [=y])
Defined at package/uclibc/Config.in:19
Depends on: <choice> && !BR2_arc [=n]
 && !BR2_avr32 [=n] && !BR2_xtensa [=n]
```

Figure 133: C Library Selection

After that, the Binutils version is selected. The GNU Binutils is a collection of binary tools to generate and manipulate binaries for a given CPU architecture. Moreover, the GNU Cross Compiler (GCC) version is also set.


Buildroot 2014.05-git-00969-ga729659 Configuration

File Edit Option Help

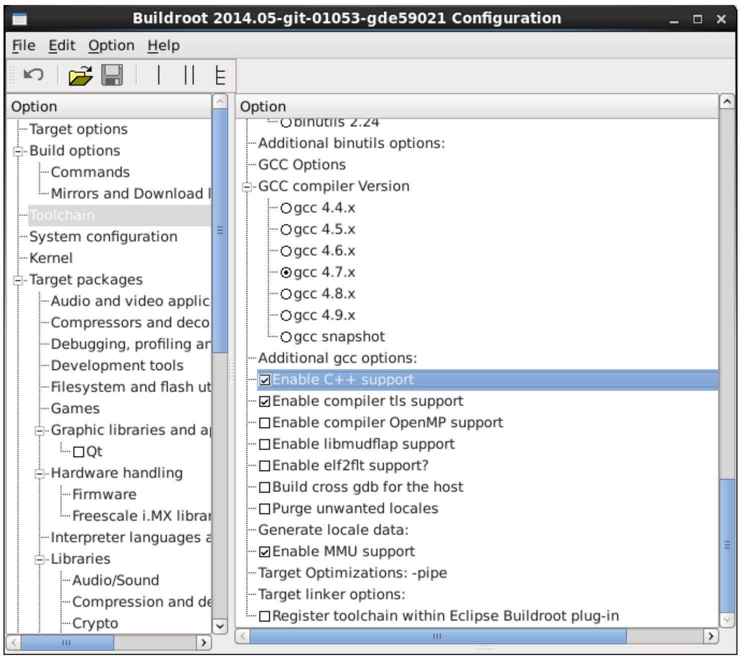
Option
Target options
Build options
Commands
Mirrors and Download locations
Toolchain
System configuration
Kernel
Target packages
Audio and video applications
Compressors and decompressor
Debugging, profiling and benchn
Development tools
Filesystem and flash utilities
Games
Graphic libraries and application
Hardware handling
Firmware
Freescale i.MX libraries
Interpreter languages and scripti
Libraries
Audio/Sound
Compression and decompres
Crypto
Database
Filesystem

Option
Binutils Version (NEW)
○ binutils 2.20.1 (NEW)
○ binutils 2.21 (NEW)
○ binutils 2.21.1 (NEW)
[✓ binutils 2.22 (NEW)]
○ binutils 2.23.1 (NEW)
○ binutils 2.23.2 (NEW)
○ binutils 2.24 (NEW)
Additional binutils options: (NEW)
GCC Options
GCC compiler Version (NEW)
○ gcc 4.4.x (NEW)
○ gcc 4.5.x (NEW)
○ gcc 4.6.x (NEW)
[✓ gcc 4.7.x (NEW)]
○ gcc 4.8.x (NEW)
○ gcc 4.9.x (NEW)
○ gcc snapshot (NEW)

```
Location:
-> Toolchain
-> uClibc C library Version (<choice> [=y])
Defined at package/uclibc/Config.in:19
Depends on: <choice> && !BR2_arc [=n]
 && !BR2_avr32 [=n] && !BR2_xtensa [=n]
```

Figure 134: Binutils Version Selection

Next, the C++ support is enabled, which offers a set of C++ libraries.


Buildroot 2014.05-git-01053-gde59021 Configuration

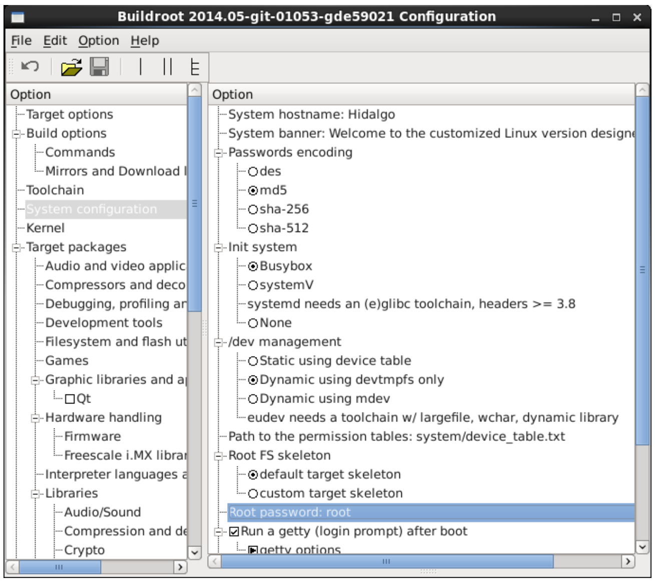
File Edit Option Help

Option
Target options
Build options
Commands
Mirrors and Download l
Toolchain
System configuration
Kernel
Target packages
Audio and video applic
Compressors and deco
Debugging, profiling ar
Development tools
Filesystem and flash ut
Games
Graphic libraries and a
☐ Qt
Hardware handling
Firmware
Freescale i.MX librar
Interpreter languages a
Libraries
Audio/Sound
Compression and de
Crypto

Option
○ binutils 2.24
Additional binutils options:
GCC Options
GCC compiler Version
○ gcc 4.4.x
○ gcc 4.5.x
○ gcc 4.6.x
[✓ gcc 4.7.x]
○ gcc 4.8.x
○ gcc 4.9.x
○ gcc snapshot
Additional gcc options:
☑ Enable C++ support
☑ Enable compiler tls support
☐ Enable compiler OpenMP support
☐ Enable libmudflap support
☐ Enable elf2flt support?
☐ Build cross gdb for the host
☐ Purge unwanted locales
Generate locale data:
☑ Enable MMU support
Target Optimizations: -pipe
Target linker options:
☐ Register toolchain within Eclipse Buildroot plug-in

Figure 135: C++ Support Enabled

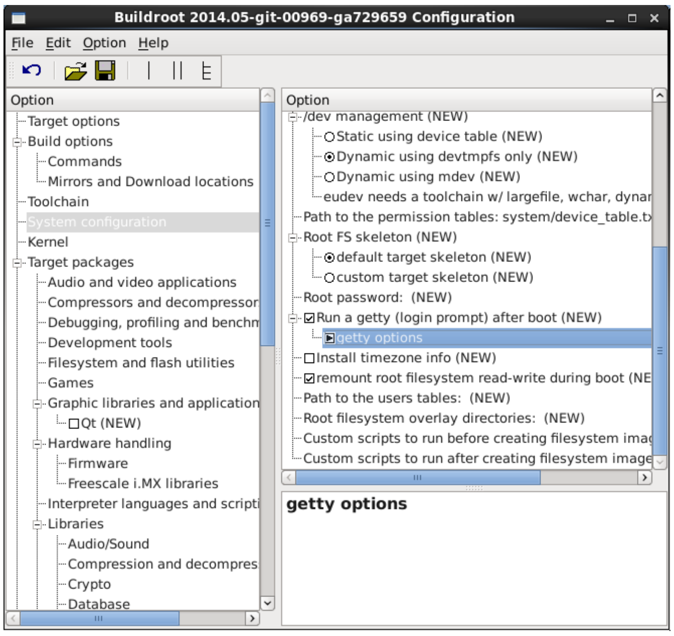
Afterward, a system hostname and system banner can be typed. In addition, the /dev management options can be set. Remember that the /dev directory contains special files which allow userspace applications to access the hardware devices by the Linux kernel. Without the device files, the user applications would not be able to use the hardware devices. On the other hand, Buildroot provides a default filesystem skeleton under the directory system/skeleton and the developer can customize it. It can be also selected the init system file and the password for the root user.



```
Option                            Option
 Target options                    System hostname: Hidalgo
 Build options                     System banner: Welcome to the customized Linux version designe
   Commands                        Passwords encoding
   Mirrors and Download l            ○des
 Toolchain                           ⦿md5
 System configuration                ○sha-256
 Kernel                              ○sha-512
 Target packages                   Init system
   Audio and video applic            ⦿Busybox
   Compressors and deco              ○systemV
   Debugging, profiling ar           systemd needs an (e)glibc toolchain, headers >= 3.8
   Development tools                 ○None
   Filesystem and flash ut         /dev management
   Games                             ○Static using device table
   Graphic libraries and a           ⦿Dynamic using devtmpfs only
     ☐Qt                             ○Dynamic using mdev
   Hardware handling                 eudev needs a toolchain w/ largefile, wchar, dynamic library
     Firmware                      Path to the permission tables: system/device_table.txt
     Freescale i.MX librai         Root FS skeleton
   Interpreter languages a           ⦿default target skeleton
   Libraries                         ○custom target skeleton
     Audio/Sound                   Root password: root
     Compression and de            ☑Run a getty (login prompt) after boot
     Crypto                          ▣getty options
```

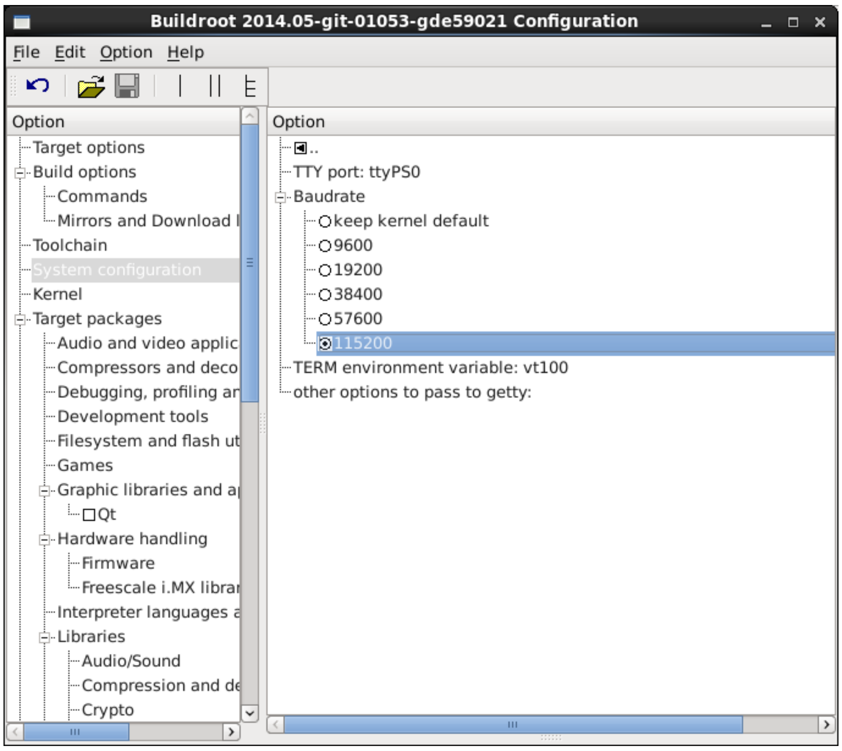
Figure 136: System Configuration Selection

Thereupon, the baud rate must be selected in order to be able to connect to an external computer and be able to communicate with it. It can be done by clicking in the “getty options”.



```
Option                                   Option
 Target options                           /dev management (NEW)
 Build options                              ○Static using device table (NEW)
   Commands                                 ⦿Dynamic using devtmpfs only (NEW)
   Mirrors and Download locations           ○Dynamic using mdev (NEW)
 Toolchain                                  eudev needs a toolchain w/ largefile, wchar, dynar
 System configuration                     Path to the permission tables: system/device_table.tx
 Kernel                                   Root FS skeleton (NEW)
 Target packages                            ⦿default target skeleton (NEW)
   Audio and video applications             ○custom target skeleton (NEW)
   Compressors and decompressor           Root password:  (NEW)
   Debugging, profiling and benchm        ☑Run a getty (login prompt) after boot (NEW)
   Development tools                        ▣getty options
   Filesystem and flash utilities         ☐Install timezone info (NEW)
   Games                                  ☑remount root filesystem read-write during boot (NE
   Graphic libraries and application      Path to the users tables:  (NEW)
     ☐Qt (NEW)                            Root filesystem overlay directories:  (NEW)
   Hardware handling                      Custom scripts to run before creating filesystem imag
     Firmware                             Custom scripts to run after creating filesystem image
     Freescale i.MX libraries
   Interpreter languages and scripti     getty options
   Libraries
     Audio/Sound
     Compression and decompres
     Crypto
     Database
```

Figure 137: Getty Options Selection



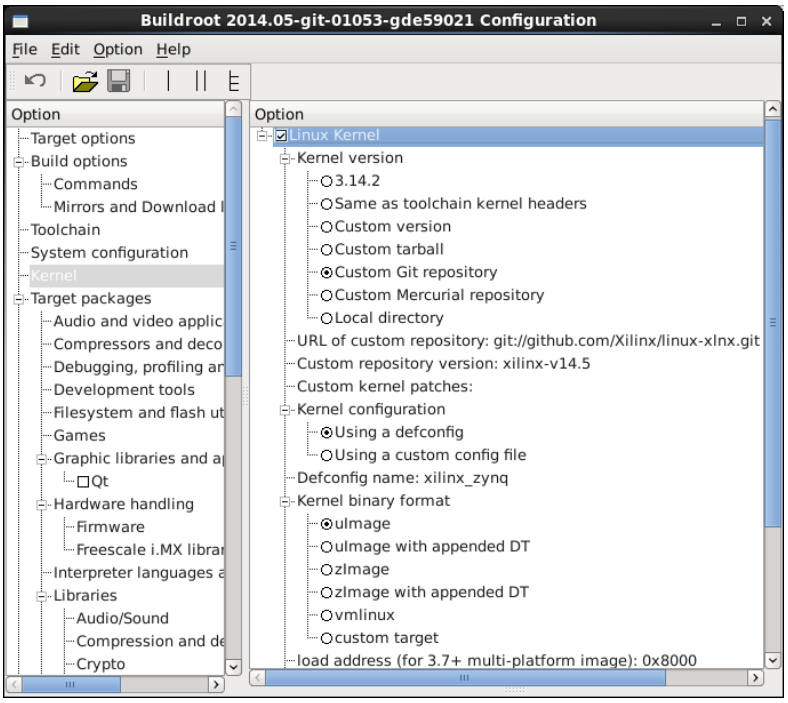
Buildroot 2014.05-git-01053-gde59021 Configuration

File Edit Option Help

```
Option                          Option
 Target options                  ..
 Build options                   TTY port: ttyPS0
   Commands                      Baudrate
   Mirrors and Download l          ○keep kernel default
 Toolchain                         ○9600
 System configuration              ○19200
 Kernel                            ○38400
 Target packages                   ○57600
   Audio and video applic          ⦿115200
   Compressors and deco          TERM environment variable: vt100
   Debugging, profiling ar       other options to pass to getty:
   Development tools
   Filesystem and flash ut
   Games
   Graphic libraries and a
     ☐Qt
   Hardware handling
     Firmware
     Freescale i.MX librar
   Interpreter languages a
   Libraries
     Audio/Sound
     Compression and de
     Crypto
```

Figure 138: Baud Rate Selection

Then, the kernel can be set. The kernel version, the kernel configuration and the kernel binary format must be selected. Note that the linux-xlnx repository will be chosen as kernel version. In addition, the “uImage” file will be set as kernel binary format.



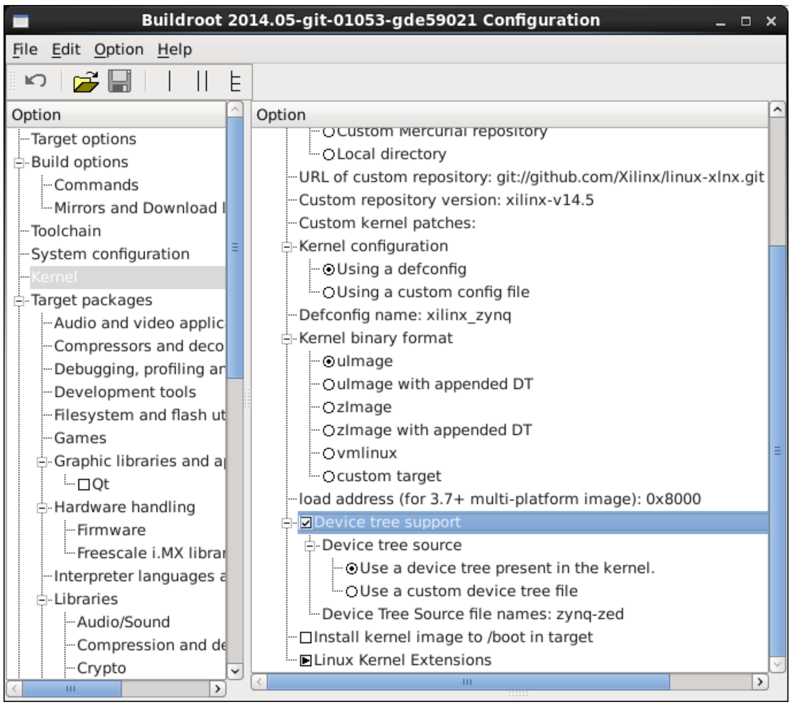
Buildroot 2014.05-git-01053-gde59021 Configuration

File Edit Option Help

```
Option                          Option
 Target options                  ☑Linux Kernel
 Build options                     Kernel version
   Commands                          ○3.14.2
   Mirrors and Download l            ○Same as toolchain kernel headers
 Toolchain                           ○Custom version
 System configuration                ○Custom tarball
 Kernel                              ⦿Custom Git repository
 Target packages                     ○Custom Mercurial repository
   Audio and video applic            ○Local directory
   Compressors and deco            URL of custom repository: git://github.com/Xilinx/linux-xlnx.git
   Debugging, profiling ar         Custom repository version: xilinx-v14.5
   Development tools               Custom kernel patches:
   Filesystem and flash ut         Kernel configuration
   Games                             ⦿Using a defconfig
   Graphic libraries and a           ○Using a custom config file
     ☐Qt                           Defconfig name: xilinx_zynq
   Hardware handling               Kernel binary format
     Firmware                        ⦿uImage
     Freescale i.MX librar           ○uImage with appended DT
   Interpreter languages a           ○zImage
   Libraries                         ○zImage with appended DT
     Audio/Sound                     ○vmlinux
     Compression and de              ○custom target
     Crypto                        load address (for 3.7+ multi-platform image): 0x8000
```

Figure 139: Kernel Setting

After that, the Device Tree support must be selected and set too. In this case, the default device present in the linux-xlnx folder will be chosen. Remember that if some custom project had done, its custom device tree file should be selected.



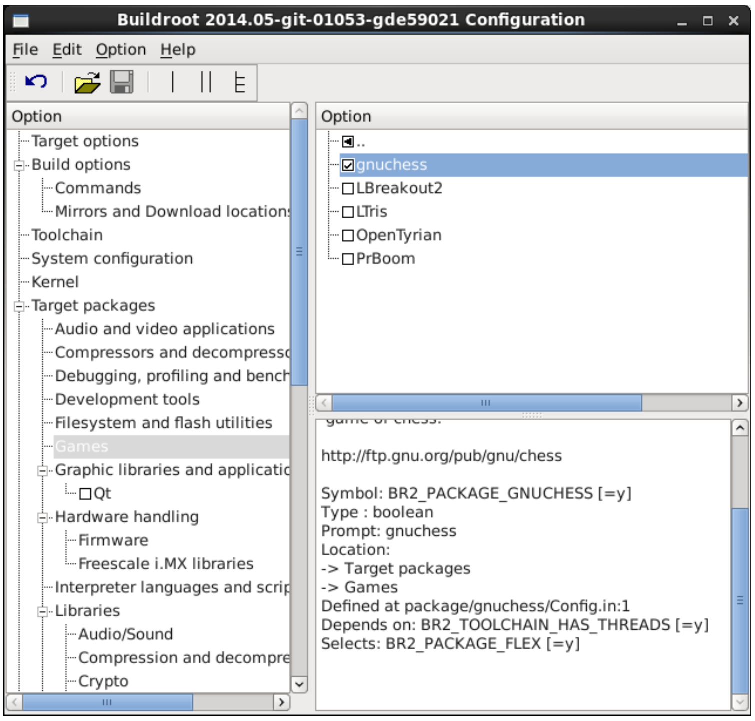
```
Buildroot 2014.05-git-01053-gde59021 Configuration
File  Edit  Option  Help

Option                              Option
 Target options                       ○Custom Mercurial repository
 Build options                        ○Local directory
   Commands                         URL of custom repository: git://github.com/Xilinx/linux-xlnx.git
   Mirrors and Download l           Custom repository version: xilinx-v14.5
 Toolchain                          Custom kernel patches:
 System configuration               Kernel configuration
 Kernel                               ●Using a defconfig
 Target packages                      ○Using a custom config file
   Audio and video applic           Defconfig name: xilinx_zynq
   Compressors and deco             Kernel binary format
   Debugging, profiling ar            ●uImage
   Development tools                  ○uImage with appended DT
   Filesystem and flash ut            ○zImage
   Games                              ○zImage with appended DT
   Graphic libraries and a            ○vmlinux
     ☐Qt                              ○custom target
   Hardware handling                load address (for 3.7+ multi-platform image): 0x8000
     Firmware                       ☑Device tree support
     Freescale i.MX librar            Device tree source
   Interpreter languages a              ●Use a device tree present in the kernel.
   Libraries                            ○Use a custom device tree file
     Audio/Sound                      Device Tree Source file names: zynq-zed
     Compression and de             ☐Install kernel image to /boot in target
     Crypto                         ▣Linux Kernel Extensions
```

Figure 140: Device Tree Support

Later, the target packages can be added or removed. In this case, a GNU Chess game will be added in order to be able to test the Operating System on ZedBoard. The rest of programs will be the programs which the default configuration includes. Anyway, it is important to know the great range of **available programs which can be added**:

- Audio and video applications.
- Compressors and decompressors.
- Debugging, profiling and benchmark.
- Development tools.
- Filesystem and flash utilities.
- Games.
- Graphic libraries and applications.
- Hardware handling.
- Interpreter languages and scripting.
- Libraries, such as audio/sound, compression and decompression, crypto, database, filesystem, graphics, hardware handling, javascript, networking, security, among others.
- Miscellaneous.
- Mail.
- Networking applications.
- Package managers.

- Real-Time.
- Shell and utilities.
- System tools.
- Text editors and viewers.



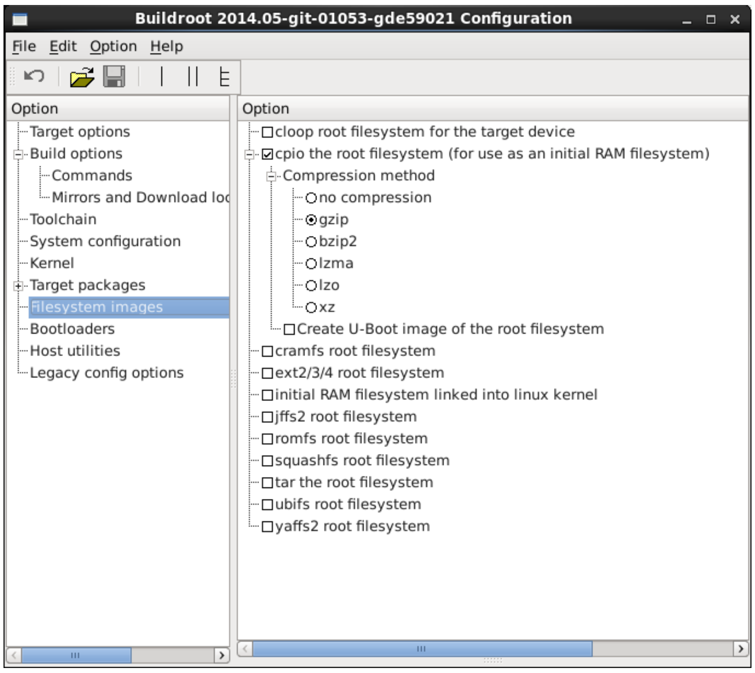
```
Option
  Target options
  Build options
    Commands
    Mirrors and Download locations
  Toolchain
  System configuration
  Kernel
  Target packages
    Audio and video applications
    Compressors and decompresso
    Debugging, profiling and bench
    Development tools
    Filesystem and flash utilities
    Games
    Graphic libraries and applicatio
      ☐Qt
    Hardware handling
      Firmware
      Freescale i.MX libraries
    Interpreter languages and scrip
    Libraries
      Audio/Sound
      Compression and decompre
      Crypto
```

```
Option
  ..
  ☑gnuchess
  ☐LBreakout2
  ☐LTris
  ☐OpenTyrian
  ☐PrBoom
```

```
http://ftp.gnu.org/pub/gnu/chess

Symbol: BR2_PACKAGE_GNUCHESS [=y]
Type : boolean
Prompt: gnuchess
Location:
-> Target packages
-> Games
Defined at package/gnuchess/Config.in:1
Depends on: BR2_TOOLCHAIN_HAS_THREADS [=y]
Selects: BR2_PACKAGE_FLEX [=y]
```

Figure 141: GNU Chess Selection

After that, the File System type can be selected. In this case, it is required a cpio root filesystem with a gzip compression method, which will be wrapped later.

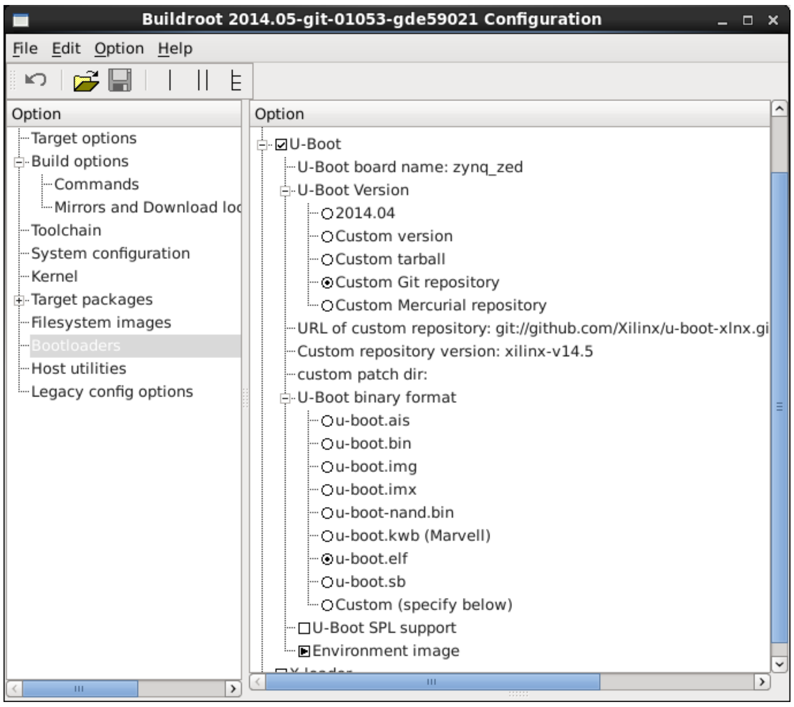


```
Option
  Target options
  Build options
    Commands
    Mirrors and Download loc
  Toolchain
  System configuration
  Kernel
  Target packages
  Filesystem images
  Bootloaders
  Host utilities
  Legacy config options
```

```
Option
  ☐cloop root filesystem for the target device
  ☑cpio the root filesystem (for use as an initial RAM filesystem)
    Compression method
      ○no compression
      ◉gzip
      ○bzip2
      ○lzma
      ○lzo
      ○xz
    ☐Create U-Boot image of the root filesystem
  ☐cramfs root filesystem
  ☐ext2/3/4 root filesystem
  ☐initial RAM filesystem linked into linux kernel
  ☐jffs2 root filesystem
  ☐romfs root filesystem
  ☐squashfs root filesystem
  ☐tar the root filesystem
  ☐ubifs root filesystem
  ☐yaffs2 root filesystem
```

Figure 142: File System File Selection

Next, the boot loader is set. In this case, U-Boot will be chosen as boot loader, selecting the u-boot-xlnx repository. In addition, the required binary format is u-boot.elf.



- ☑ U-Boot
  - U-Boot board name: zynq_zed
  - U-Boot Version
    - ○ 2014.04
    - ○ Custom version
    - ○ Custom tarball
    - ⦿ Custom Git repository
    - ○ Custom Mercurial repository
  - URL of custom repository: git://github.com/Xilinx/u-boot-xlnx.gi
  - Custom repository version: xilinx-v14.5
  - custom patch dir:
  - U-Boot binary format
    - ○ u-boot.ais
    - ○ u-boot.bin
    - ○ u-boot.img
    - ○ u-boot.imx
    - ○ u-boot-nand.bin
    - ○ u-boot.kwb (Marvell)
    - ⦿ u-boot.elf
    - ○ u-boot.sb
    - ○ Custom (specify below)
  - ☐ U-Boot SPL support
  - ▣ Environment image

Figure 143: Boot Loader Settings

Afterward, some host utilities or legacy config options could be selected. Nevertheless, it will not be necessary in this case. Therefore, this configuration can be saved by clicking “Ctrl” + “S”, and exit by clicking “File” and “Quit”.

Finally, once the settings have been properly configured, BuildRoot launches the file generation process for the ZedBoard by typing the following command:

```
make
```

Command Window 12: Building the Required Files for ZedBoard

In BuildRoot, the make command performs the following steps:

- Download the selected source files and repositories.
- Configure, build and install the cross-compiling toolchain.
- Build and install the selected target packages.
- Build the kernel file in the selected format.
- Create the root filesystem in the selected configuration.

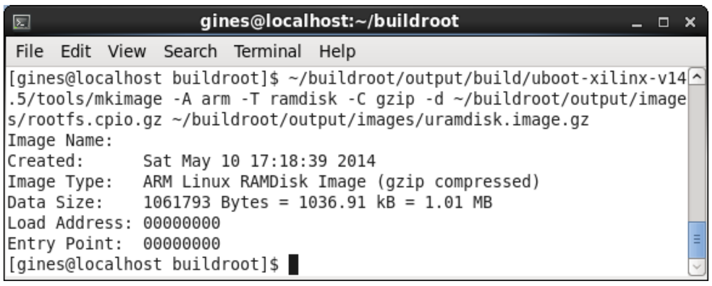
The “u-boot”, “zynq-zed.dtb”, “uImage” and “rootfs.cpio.gz” files has been built and copied into the following folder.

…/buildroot/output/images

Nevertheless, the ramdisk image ("rootfs.cpio.gz") has to be wrapped with a U-Boot wrapper using the mkimage utility available in the U-Boot directory. Therefore, the last command to be executed is the following:

```
    ~/buildroot/output/build/uboot-xilinx-v14.5/tools/mkimage   -A
arm -T ramdisk -C gzip -d ~/buildroot/output/images/rootfs.cpio.gz
~/buildroot/output/images/uramdisk.image.gz
```

Command Window 13: Wrapping the RamDisk File into an uRamDisk File

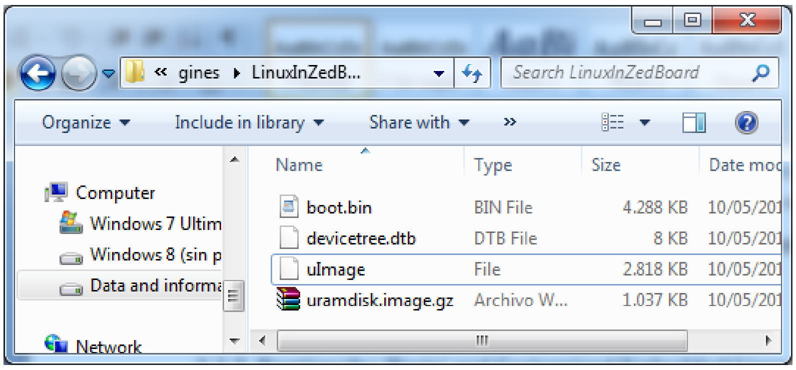

gines@localhost:~/buildroot
File Edit View Search Terminal Help

```
[gines@localhost buildroot]$ ~/buildroot/output/build/uboot-xilinx-v14
.5/tools/mkimage -A arm -T ramdisk -C gzip -d ~/buildroot/output/image
s/rootfs.cpio.gz ~/buildroot/output/images/uramdisk.image.gz
Image Name:
Created:      Sat May 10 17:18:39 2014
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    1061793 Bytes = 1036.91 kB = 1.01 MB
Load Address: 00000000
Entry Point:  00000000
[gines@localhost buildroot]$
```

Figure 144: Wrapping the RamDisk File into an uRamDisk File

In addition, "zynq-zed.dtb" has to be renamed as "devicetree.dtb" and "u-boot" as "u-boot.elf". Finally, the files are ready to be copied into Windows. After copying the U-Boot file has to be copy into the "boot.bin" folder, the subchapter "3.3.3. Boot Image, boot.bin" can be finished.

After that, the rest of the files can be copied into the "LinuxInZedboard" folder.



Name	Type	Size	Date mod
boot.bin	BIN File	4.288 KB	10/05/201
devicetree.dtb	DTB File	8 KB	10/05/201
uImage	File	2.818 KB	10/05/201
uramdisk.image.gz	Archivo W...	1.037 KB	10/05/201

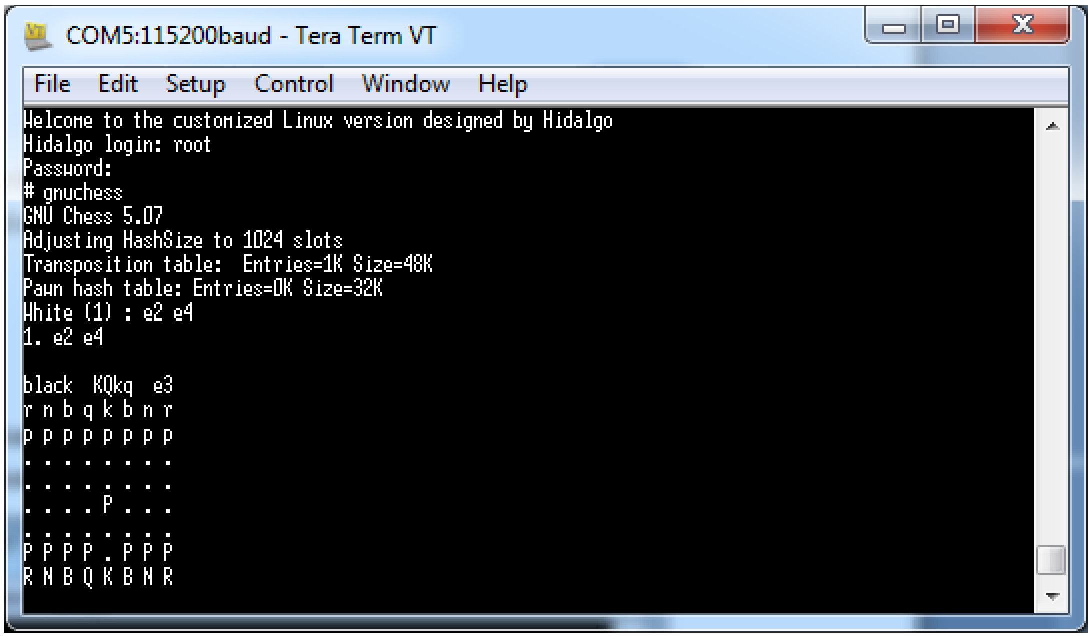
Figure 145: LinuxInZedBoard Folder

Finally, remember that Buildroot provides several ways of extra customization if a more custom configuration is required.

- Extra filesystem configuration by customizing the BuildRoot skeleton.
- Extra µClibc configuration by typing "make uclibc-menuconfig".
- Extra Linux-xlnx configuration by typing "make linux-menuconfig".
- Extra BusyBox configuration by typing "make busybox-menuconfig".
- Extra BareBox configuration by typing "make barebox-menuconfig".

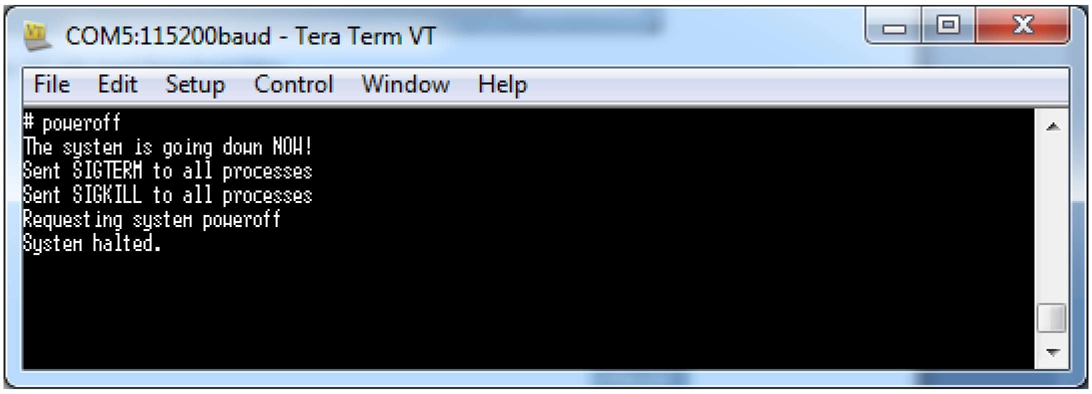
2.10.2. TeraTerm

These four files are the required files that should be copied into the SD Card in order to be able to run this Linux OS on ZedBoard. Therefore, the ZedBoard can be switch on. Remember that the user name and password are "root". The GNU Chess game can be launched by typing "gnuchess". Finally, the "poweroff" will shut down the Operating System.


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
Welcome to the customized Linux version designed by Hidalgo
Hidalgo login: root
Password:
# gnuchess
GNU Chess 5.07
Adjusting HashSize to 1024 slots
Transposition table:  Entries=1K Size=48K
Pawn hash table: Entries=0K Size=32K
White (1) : e2 e4
1. e2 e4

black  KQkq  e3
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . P . . .
. . . . . . . .
P P P P . P P P
R N B Q K B N R
```

Figure 146: GNUChess Displaying


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
# poweroff
The system is going down NOW!
Sent SIGTERM to all processes
Sent SIGKILL to all processes
Requesting system poweroff
System halted.
```

Figure 147: Shutting Down the Operating System

3. Ubuntu Linux OS on ZedBoard

3.1. Overview

As already mentioned in the third and fourth chapter, the aim is to build a Linaro-Ubuntu Embedded Linux OS on ZedBoard. For this purpose, the required SD card partitions and files are showed bellow:

1. **FAT32** partition:
    a. The **Boot Image** ("boot.bin"), which requires the same three components:
        d. The **FSBL** (First Stage Boot Loader).
        e. The **Programmable Logic Hardware BitStream** (optional).
        f. The **U-Boot** (Second Stage Boot Loader).
    b. The **Device Tree Binary File** ("devicetree.dtb"), which is again obtained from the Device Tree Source File and loaded into the DDR memory by U-Boot.
    c. The **Linux Kernel File** ("uImage"), which is again loaded into the DDR memory by U-Boot.
2. **ext4** partition:
    a. The **Root File System Image**, which is also loaded into the DDR memory by U-Boot. It contains the Operating System itself. Nevertheless, it will not be a compress file in this case.

Therefore, the programs and steps to create, configure, build and implement this Ubuntu Linux OS version on ZedBoard are summarized below.

1. **VMware Player:**
    a. Preparing the SD Card.
2. **Xilinx Platform Studio (XPS):**
    a. Downloading and loading the Project.
    b. Generating the BitStream File and exporting to SDK.
3. **Software Development Kit (SDK):**
    a. Creating the FSBL (First Stage Boot Loader).
4. **VMware Player:**
    a. Configuring and obtaining the U-Boot file.
5. **Software Development Kit (SDK):**
    a. Creating the boot image ("boot.bin") from the FSBL, the BitStream file and the U-Boot file.
6. **VMware Player:**
    a. Configuring and obtaining the Device Tree Binary file.
    b. Configuring and obtaining the Kernel Image.
7. **SD Card:**
    a. Copying the files in the SD Card.
8. **Tera Term:**
    a. Running Linaro-Ubuntu Linux on ZedBoard.

3.2. Preparing the SD Card

3.2.1. VMware Player

The first step is to prepare the SD Card, which will be prepared by using the GParted Partition Editor program. Two empty partitions will be created with GParted. The first partition will be a 52 MB FAT32 format, and the second one will use the remaining space in ext4 format.

Therefore, the SD Card must be connected to the laptop or PC and connect to the virtual machine by clicking with the right mouse button in "Connect (Disconnect from host)" in the SD Card icon. In this case, the icon is in the highest and rightmost position.

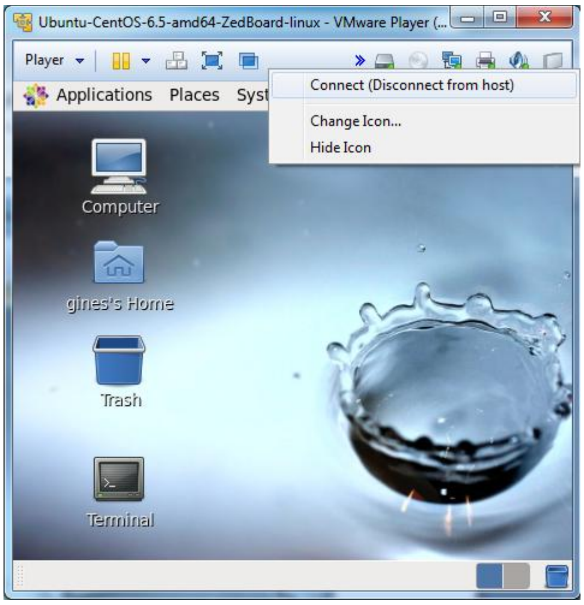

Ubuntu-CentOS-6.5-amd64-ZedBoard-linux - VMware Player (...
Player
Applications
Places
Syst
Connect (Disconnect from host)
Change Icon...
Hide Icon
Computer
gines's Home
Trash
Terminal

Figure 148: SD Card Conection to the VMware Player

GParted must be opened in "Applications", "System Tools", "GParted Partition Editor". It will also ask for the root password.

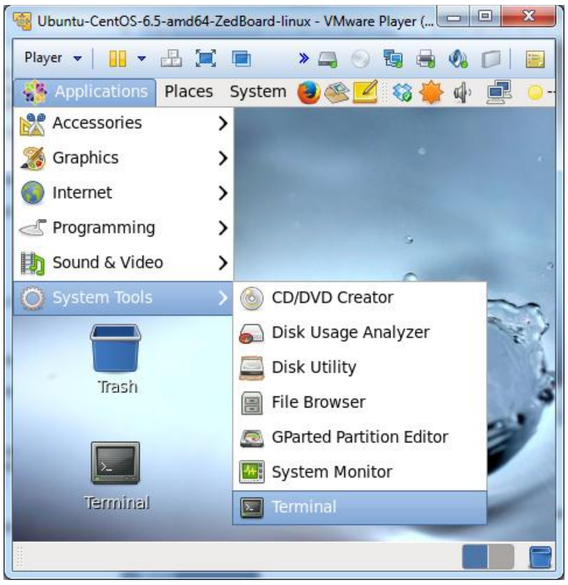
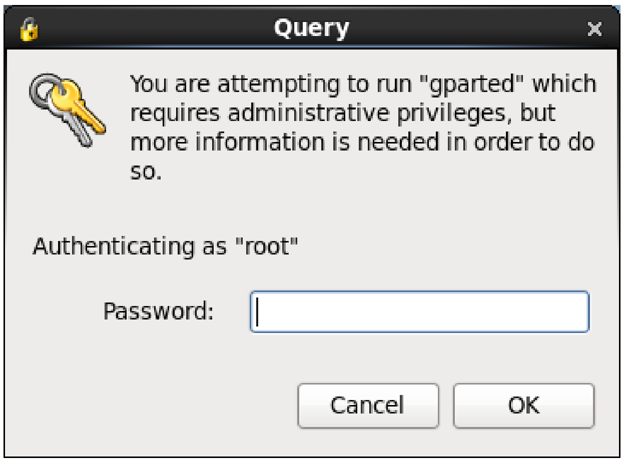


Figure 149: Opening GParted Parition Editor



Query

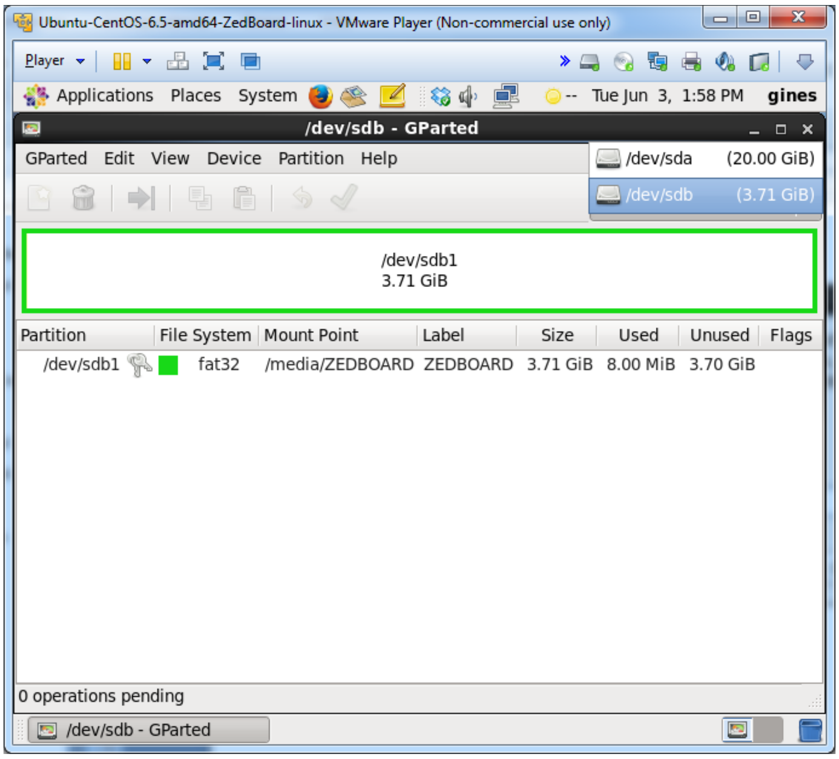
You are attempting to run "gparted" which requires administrative privileges, but more information is needed in order to do so.

Authenticating as "root"

Password:

Figure 150: Authentication as Root User

The SD Card must be selected in the up-right menu.



Ubuntu-CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)

Player
Applications Places System
Tue Jun 3, 1:58 PM gines
/dev/sdb - GParted
GParted Edit View Device Partition Help

/dev/sda (20.00 GiB)
/dev/sdb (3.71 GiB)

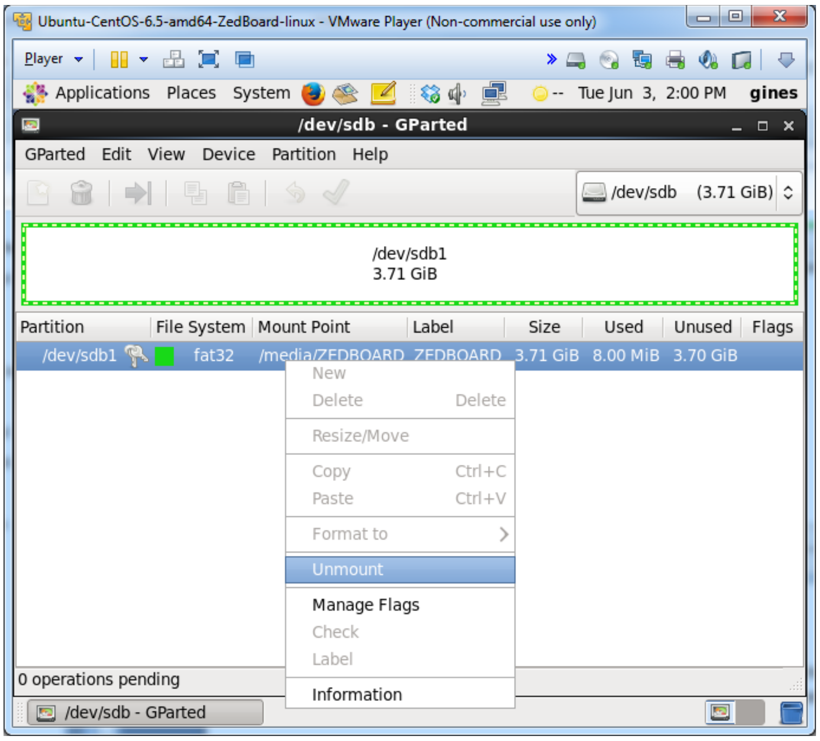
/dev/sdb1
3.71 GiB

Partition	File System	Mount Point	Label	Size	Used	Unused	Flags
/dev/sdb1	🟩 fat32	/media/ZEDBOARD	ZEDBOARD	3.71 GiB	8.00 MiB	3.70 GiB	

0 operations pending

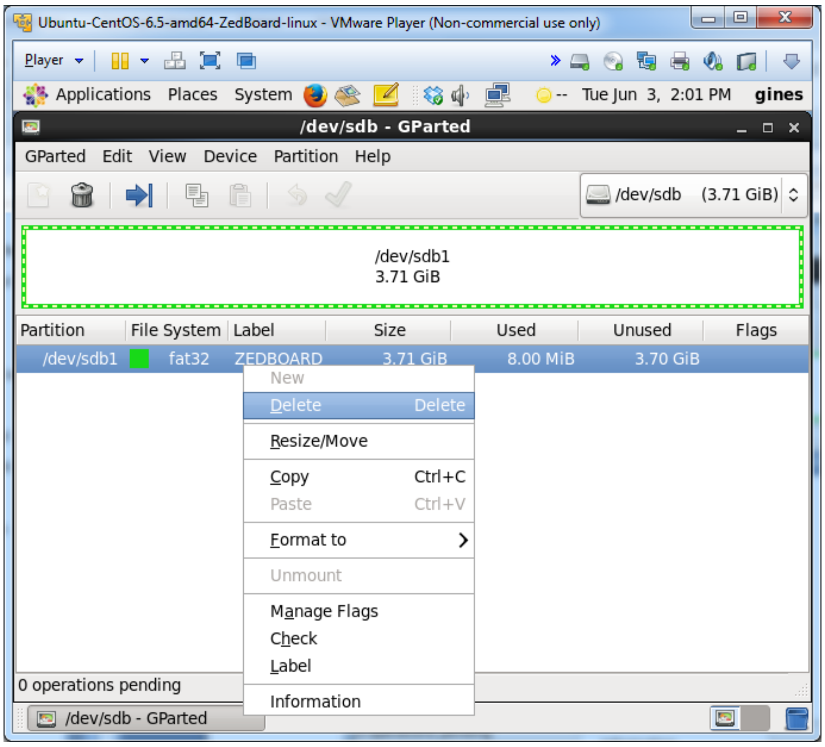
Figure 151: SD Card Selection

If the SD Card has already any partition mounted, delete them by right-clicking in each one and “Unmount”, and once again right-dicking and “Delete”.



Partition	File System	Mount Point	Label	Size	Used	Unused	Flags
/dev/sdb1	fat32	/media/ZEDBOARD	ZEDBOARD	3.71 GiB	8.00 MiB	3.70 GiB	

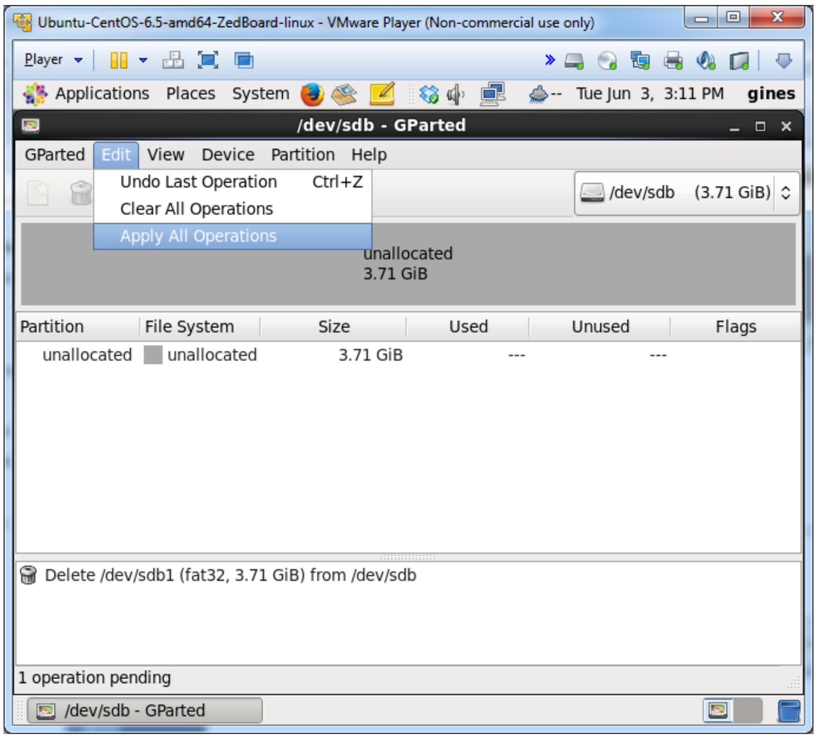
Figure 152: Unmounting Previous Partitions



Partition	File System	Label	Size	Used	Unused	Flags
/dev/sdb1	fat32	ZEDBOARD	3.71 GiB	8.00 MiB	3.70 GiB	

Figure 153: Deleting Previous Partitions

The SD Card must look empty. Then, click in “Edit”, “Apply All Operations”.


Ubuntu-CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Player
Applications Places System
Tue Jun 3, 3:11 PM gines
/dev/sdb - GParted
GParted Edit View Device Partition Help
Undo Last Operation Ctrl+Z
Clear All Operations
Apply All Operations
/dev/sdb (3.71 GiB)
unallocated
3.71 GiB
Partition File System Size Used Unused Flags
unallocated unallocated 3.71 GiB --- ---
/dev/sdb - GParted

Delete /dev/sdb1 (fat32, 3.71 GiB) from /dev/sdb

1 operation pending

Figure 154: SD Card Empty

Once the operations have completed, the message indicating that all operations were successful can be closed. After that, right-click in the “unallocated” memory of the SD Card and “New”.

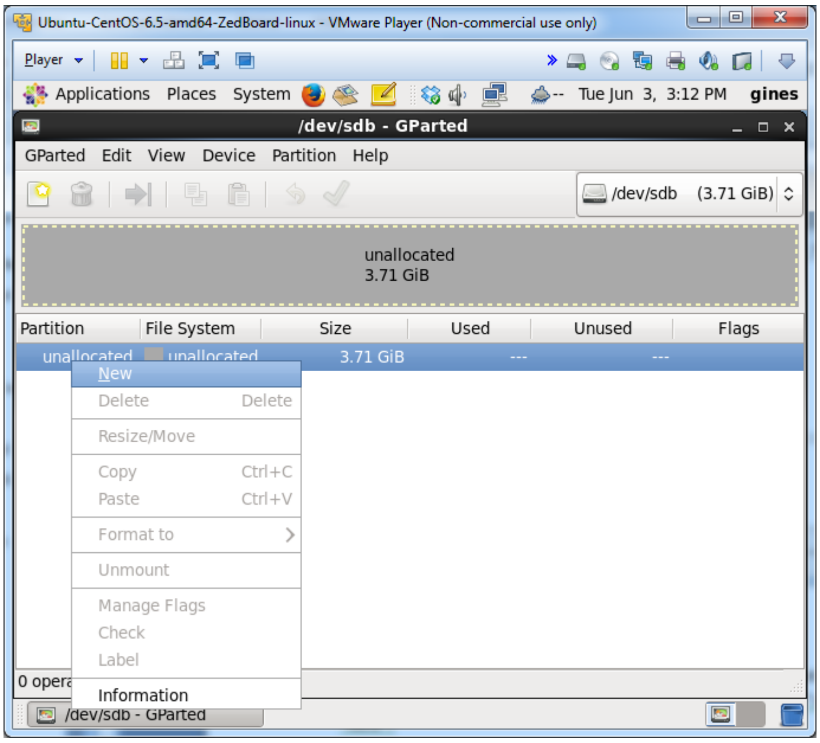
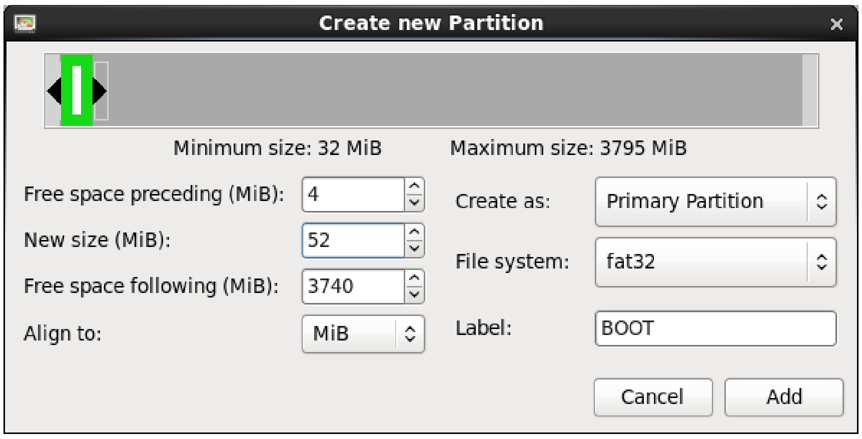

Ubuntu-CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Player
Applications Places System
Tue Jun 3, 3:12 PM gines
/dev/sdb - GParted
GParted Edit View Device Partition Help
/dev/sdb (3.71 GiB)
unallocated
3.71 GiB
Partition File System Size Used Unused Flags
unallocated unallocated 3.71 GiB --- ---
New
Delete Delete
Resize/Move
Copy Ctrl+C
Paste Ctrl+V
Format to
Unmount
Manage Flags
Check
Label
Information
0 opera
/dev/sdb - GParted

Figure 155: New Partition Creation

A new 52MB FAT32 partition must be created by filling the "Create new Partition" window as following and "Add":

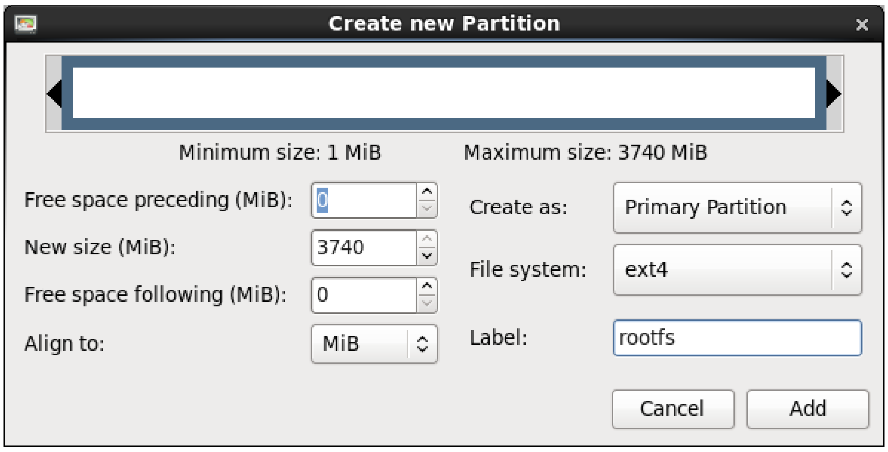


Minimum size: 32 MiB
Maximum size: 3795 MiB

Free space preceding (MiB): 4
New size (MiB): 52
Free space following (MiB): 3740
Align to: MiB
Create as: Primary Partition
File system: fat32
Label: BOOT

Figure 156: New 52MB FAT32 Partition

Create another partition, again right-button on the unallocated space and "New", and filling again as the following image to create an ext4 partition in the rest of the SD Card.



Minimum size: 1 MiB
Maximum size: 3740 MiB

Free space preceding (MiB): 0
New size (MiB): 3740
Free space following (MiB): 0
Align to: MiB
Create as: Primary Partition
File system: ext4
Label: rootfs

Figure 157: New 3740MB ext4 Partition

Once again, click "Edit" and "Apply All Operations" to save the changes. The SD Card is now formatted in the required format to boot Ubuntu on ZedBoard. Therefore, GParted Partition Editor can be closed; it will not be further necessary.

Note that if the SD Card is open with Windows, only the 52MB partition will be displayed, because Windows does not read the ext4 format. Therefore, if the SD Card has to be formatted, for instance, after finishing this guide, it will have to be done using GParted in Linux.

Finally, the SD Card can be disconnected from the virtual machine by clicking with the right mouse button in "Disconnect (Connect to host)" in the SD Card icon. Remember that this icon is in the highest and rightmost position.

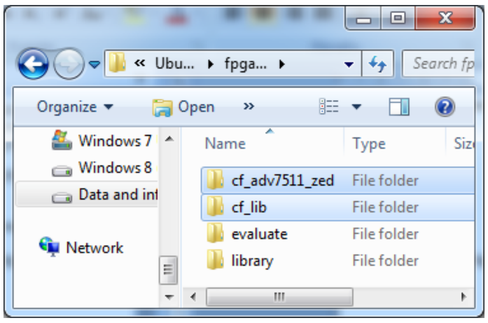
3.3. Downloading and Loading the Project

3.3.1. Xilinx Platform Studio (XPS)

Once the SD Card is ready to be used, the next step is to download the Zed HDL Reference Design from the following links:

- https://github.com/analogdevicesinc/fpgahdl_xilinx/archive/master.zip
- https://github.com/analogdevicesinc/no-OS/archive/master.zip

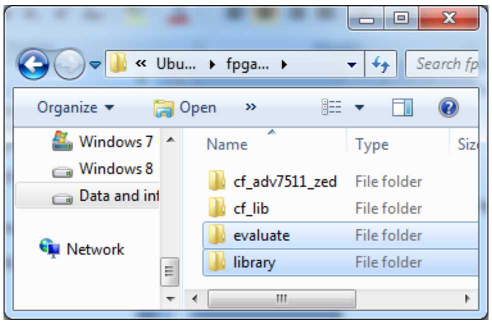
After unzipping the downloaded "fpgahdl_xilinx-master.zip" file, only these two highlighted files showed below are required; the rest can be deleted.



Name	Type
cf_adv7511_zed	File folder
cf_lib	File folder
evaluate	File folder
library	File folder

Figure 158: "fpgahdl_xilinx-master.zip" Required Files

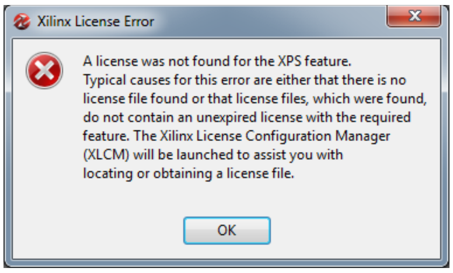
On the other hand, the required files from the "no-OS-master.zip" file are located in the "adv7511" subfolder, and they are showed below:



Name	Type
cf_adv7511_zed	File folder
cf_lib	File folder
evaluate	File folder
library	File folder

Figure 159: "no-OS-master.zip" Required Files

The "system.xmp" file is available in "cf_adv7511_zed" and can be open. This will launch Xilinx Platform Studio. After XPS is opened, an error is seen with the WebPACK license in Xilinx 14.7, this is a bug. WebPACK includes the Zynq XPS license; simply click "OK" and Close the Xilinx License Configuration Manager after it opens:



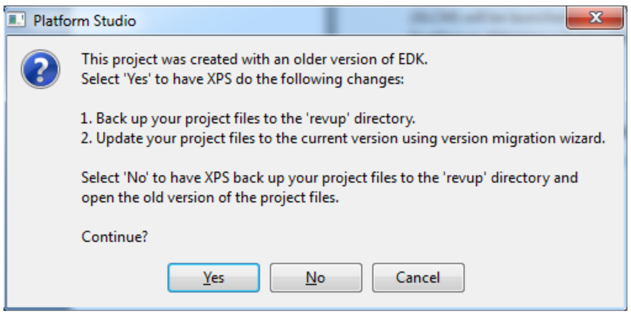
Xilinx License Error

A license was not found for the XPS feature. Typical causes for this error are either that there is no license file found or that license files, which were found, do not contain an unexpired license with the required feature. The Xilinx License Configuration Manager (XLCM) will be launched to assist you with locating or obtaining a license file.

OK

Figure 160: Xilinx License Bug

This project has been done in a previous Xilinx version, in the 14.6 version. Nevertheless, Xilinx can automatically update the project to the current version. Therefore, click in "Yes" when the next message appears. The project will automatically update to the current XPS release.



This project was created with an older version of EDK.
Select 'Yes' to have XPS do the following changes:

1. Back up your project files to the 'revup' directory.
2. Update your project files to the current version using version migration wizard.

Select 'No' to have XPS back up your project files to the 'revup' directory and open the old version of the project files.

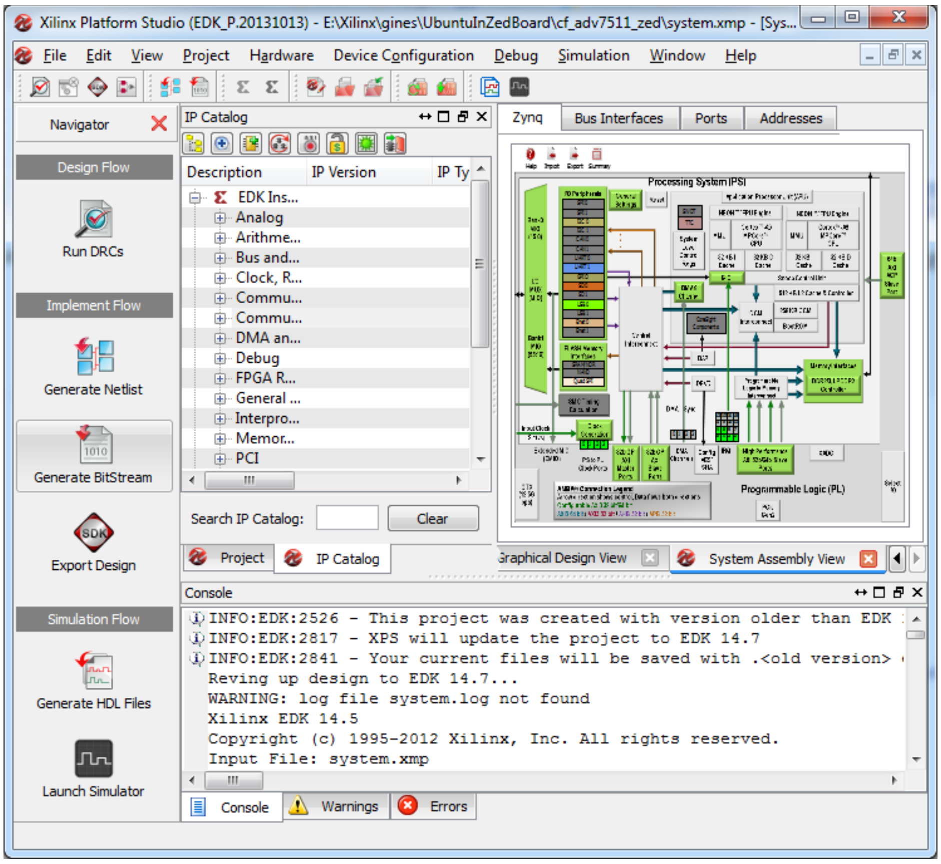
Continue?

Figure 161: Updating the Project to the Current Version

3.4. Generating the BitStream File and Exporting to SDK

3.4.1. Xilinx Platform Studio (XPS)

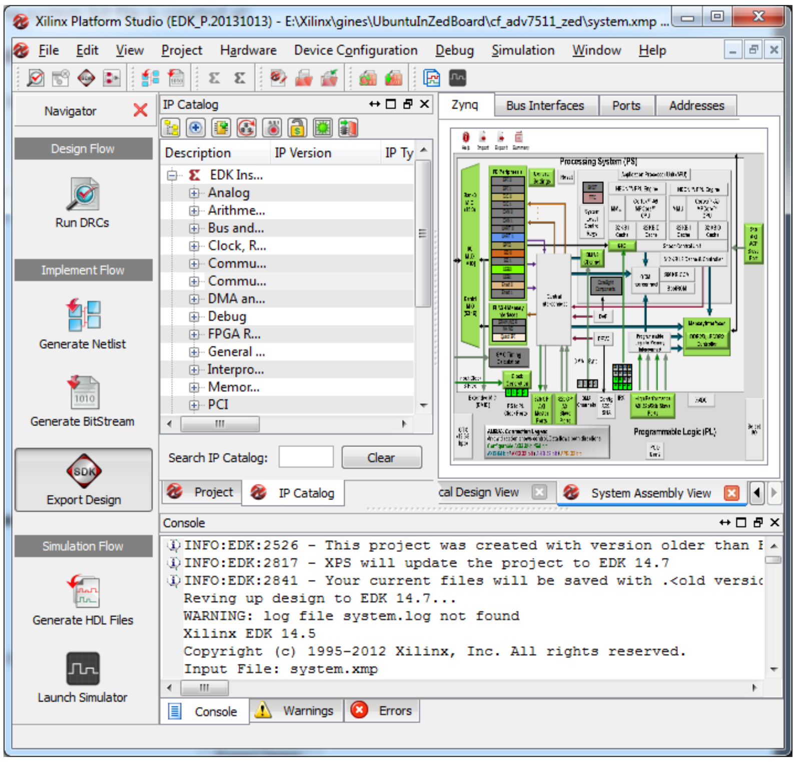
Now, the project is prepared for the current Xilinx version. Next, click on the Generate BitStream icon in the Navigator panel on the left side of XPS. The build process will create the "system.bit" file, but it will take approximately from 30 to 90 minutes depending on the PC's capabilities.



```
INFO:EDK:2526 - This project was created with version older than EDK
INFO:EDK:2817 - XPS will update the project to EDK 14.7
INFO:EDK:2841 - Your current files will be saved with .<old version>
Reving up design to EDK 14.7...
WARNING: log file system.log not found
Xilinx EDK 14.5
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.
Input File: system.xmp
```

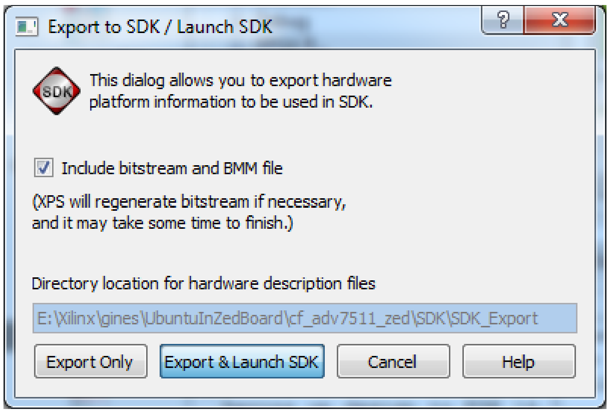
Figure 162: XPS before Generating the BitStream File

After, the design has to be exported to SDK by clicking in "Export Design" in the navigator panel. Accept the default location for the hardware description files and click in "Export & Launch SDK".



```
INFO:EDK:2526 - This project was created with version older than I
INFO:EDK:2817 - XPS will update the project to EDK 14.7
INFO:EDK:2841 - Your current files will be saved with .<old versic
Reving up design to EDK 14.7...
WARNING: log file system.log not found
Xilinx EDK 14.5
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.
Input File: system.xmp
```

Figure 163: Export Design Selection



Export to SDK / Launch SDK

This dialog allows you to export hardware platform information to be used in SDK.

[✓] Include bitstream and BMM file

(XPS will regenerate bitstream if necessary, and it may take some time to finish.)

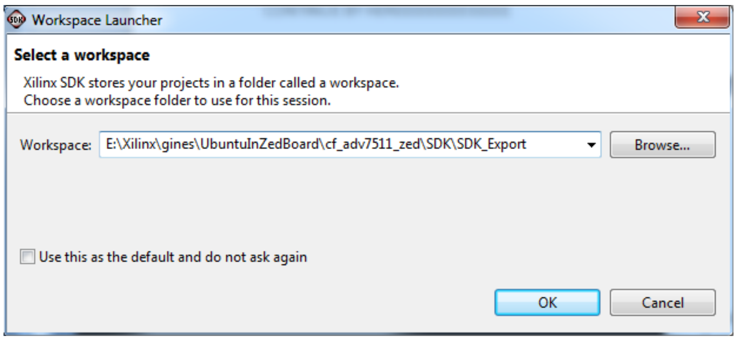
Directory location for hardware description files

E:\Xilinx\gines\UbuntuInZedBoard\cf_adv7511_zed\SDK\SDK_Export

Figure 164: Export Design to SDK

When SDK is launched, it will ask for the workspace folder. Select the folder in which XPS has generated the SDK file. This folder is the following.

…\cf_adv7511_zed\SDK\SDK_Export\SDK\SDK_Export


Workspace Launcher

Select a workspace

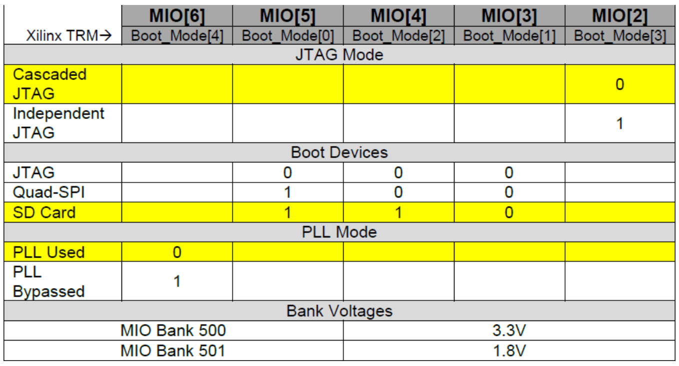
Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: E:\Xilinx\gines\UbuntuInZedBoard\cf_adv7511_zed\SDK\SDK_Export

Figure 165: Workspace Selection

3.5. Stage 0 or BootROM

As already occurred with the other Linux OS, it will be introduced on ZedBoard using the SD Card, hence the MIO Configuration Modes are the same as it was listed for the other Linux OS:



	MIO[6]	MIO[5]	MIO[4]	MIO[3]	MIO[2]
Xilinx TRM→	Boot_Mode[4]	Boot_Mode[0]	Boot_Mode[2]	Boot_Mode[1]	Boot_Mode[3]
			JTAG Mode		
Cascaded JTAG					0
Independent JTAG					1
			Boot Devices		
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
			PLL Mode		
PLL Used	0				
PLL Bypassed	1				
			Bank Voltages		
MIO Bank 500			3.3V		
MIO Bank 501			1.8V		

Figure 166: ZedBoard Configuration Modes

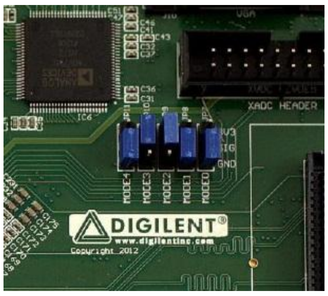


Figure 167: ZedBoard SD Card Boot Mode Jumper Setting

3.6. Boot Image, "boot.bin"

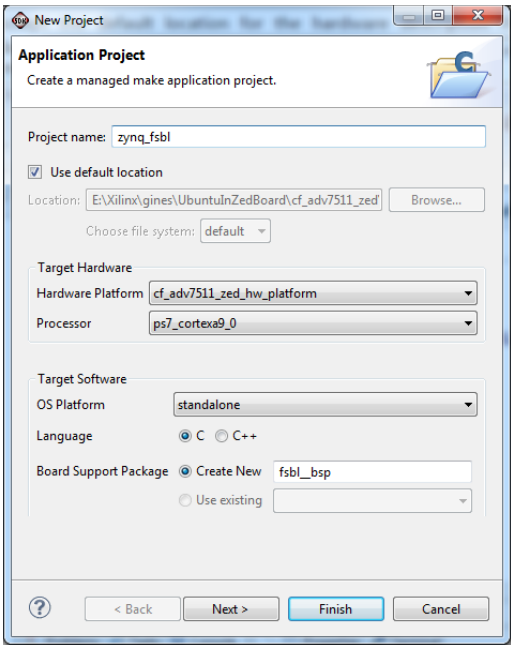
Once again, the FSBL is composed by 3 files:

- The specific First Stage Boot Loader (FSBL) for ZedBoard.
- The specific Programmable Logic Hardware BitStream file (optional).
- The U-Boot file, as Second Stage Boot Loader.

3.6.1. First Stage Boot Loader (FSBL) and BitStream

3.6.1.1. Software Development Kit (SDK)

The First Stage Boot Loader has to be built like it occurred with the previous Linux OS. For this purpose, the same steps will be repeated; click in “File”, “New” and “Application Project”. Fill the “New Project” window as the following figure, click in “Next”, select the “Zynq FSBL” option, and click in “Finish”, like it occurred with the previous Linux.


New Project

Application Project

Create a managed make application project.

Project name: zynq_fsbl

☑ Use default location

Location: E:\Xilinx\gines\UbuntuInZedBoard\cf_adv7511_zed

Choose file system: default

Target Hardware

Hardware Platform: cf_adv7511_zed_hw_platform

Processor: ps7_cortexa9_0

Target Software

OS Platform: standalone

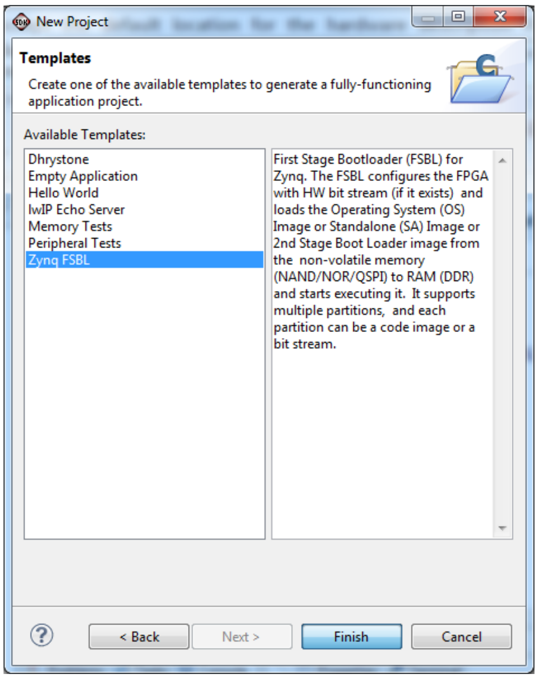
Language: [✓ C] C++

Board Support Package: [✓ Create New] fsbl_bsp

Use existing

Figure 168: Setting the FSBL for Ubuntu ½

Once the FSBL is built, it can be copied into the “UbuntuLinuxInZedBoard” folder.


New Project

Templates

Create one of the available templates to generate a fully-functioning application project.

Available Templates:

Dhrystone
Empty Application
Hello World
lwIP Echo Server
Memory Tests
Peripheral Tests
[✓ Zynq FSBL]

First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Figure 169: Setting the FSBL for Ubuntu 2/2

Copy the FSBL file, which can be directly copied from the SDK panel; and paste it into a new folder, for instance, "UbuntuLinuxInZedBoard". The "zynq_fsbl.elf" file can be located in "zynq_fsbl", "Debug".

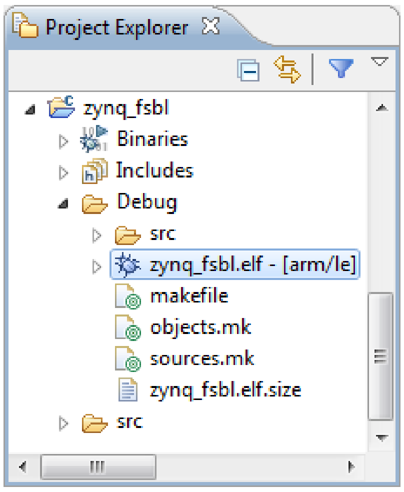
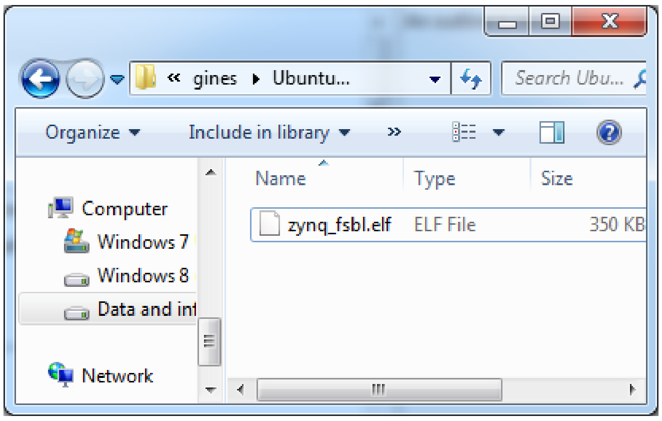


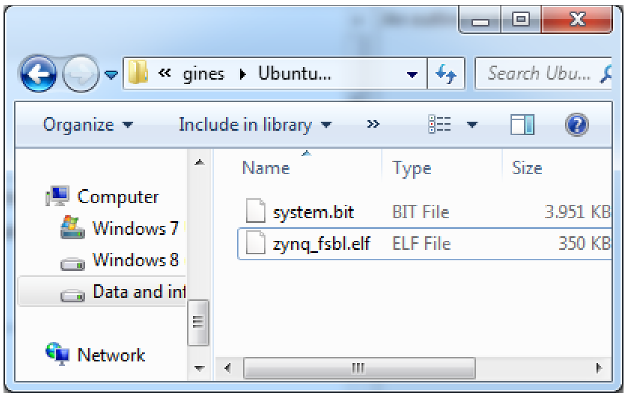
Figure 170: FSBL Location



Name	Type	Size
zynq_fsbl.elf	ELF File	350 KB

Figure 171: FSBL File Copied into UbuntuLinuxInZedBoard

The BitStream file has to be also copied into the "UbuntuLinuxInZedboard" folder. It can be also copied from the SDK panel.



Name	Type	Size
system.bit	BIT File	3.951 KB
zynq_fsbl.elf	ELF File	350 KB

Figure 172: BitStream File Copied into UbuntuLinuxInZedBoard

3.6.2. Second Stage Boot Loader, U-Boot

3.6.2.1. VMware Player

Almost the same steps followed in "2.6.2.1. First Method (optional) – U-Boot Repository" will be repeated. Nevertheless, it must be configured in order to load the Root File System Image from the ext4 partition of the SD card, unlike the previous case.

Open the Zynq-parameter file ("zynq-common.h"), located in the path showed below, in order to edit the SD card boot ("sdboot") configuration line which copies the root file system from the SD card to memory.

.../u-boot-xlnx/include/configs/

By default, it is configured to load the "uramdisk.image.gz" file, like in the previous chapter, being defined to load the Linux Kernel image, the Device Tree Binary file and the compress Root File System image. Look the following commands (search for "sdboot" to locate them).

```
"sdboot=if mmcinfo; then " \
      "run uenvboot; " \
      "echo Copying Linux from SD to RAM... && " \
      "fatload mmc 0 0x3000000 ${kernel_image} && " \
      "fatload mmc 0 0x2A00000 ${devicetree_image} && " \
      "fatload mmc 0 0x2000000 ${ramdisk_image} && " \
      "bootm 0x3000000 0x2000000 0x2A00000; " \
```

Command Window 14: Default SD Card Boot Configuration on Embedded Linux

And change them for the following lines:

```
"sdboot=echo Copying Linux kernel from SD to RAM...RFS in ext4;" \
      "mmcinfo;" \
      "fatload mmc 0 0x3000000 ${kernel_image};" \
      "fatload mmc 0 0x2A00000 ${devicetree_image};" \
      "bootm 0x3000000 - 0x2A00000\0" \
```

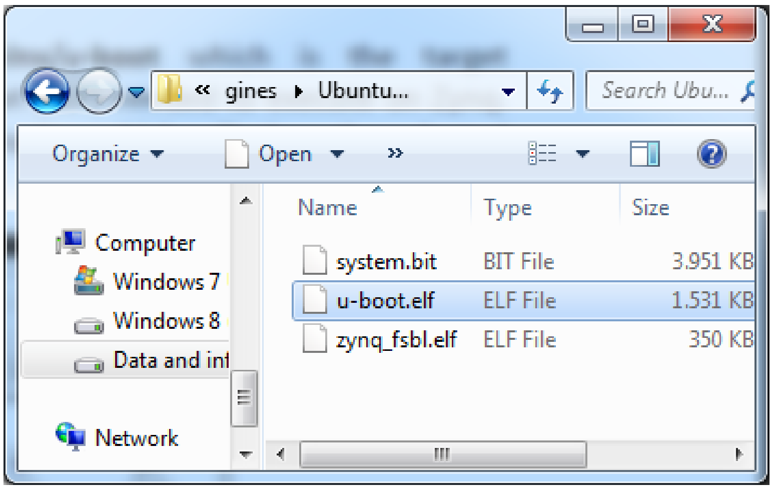
Command Window 15: New SD Card Boot Configuration

Now, repeat the configuration already done with the previous Linux OS version U-Boot will be generated by opening the "Terminal" program and typing the following commands:

```
cd ~
source .bash_profile
cd ~/u-boot-xlnx/
make distclean
make zynq_zed_config
make
```

Command Window 16: Building U-Boot

The U-Boot file will be generated as "u-boot" in the u-boot-xlnx folder. It can be copied into the "UbuntuLinuxInZedBoard" folder and renamed as "u-boot.elf".



Name	Type	Size
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
zynq_fsbl.elf	ELF File	350 KB

Figure 173: U-Boot File Copied into UbuntuLinuxInZedBoard

3.6.3. Boot Image, "boot.bin"

2.6.3.1. Software Development Kit (SDK)

The boot image will be created in SDK using the FSBL, the hardware BitStream file and the U-Boot ELF file. Whether SDK was closed, it must be re-launched with the previous workspace. Once opened, click in "Xilinx Tools" and "Create Zynq Boot Image":

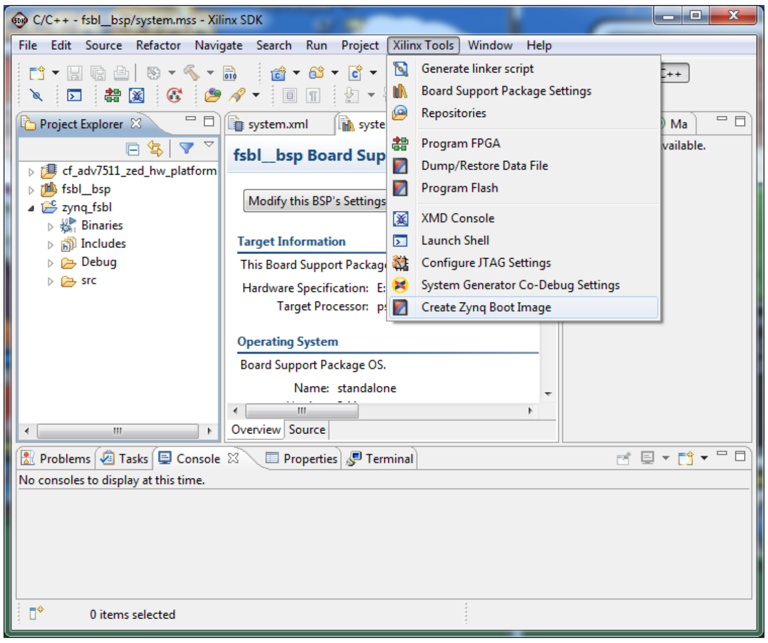
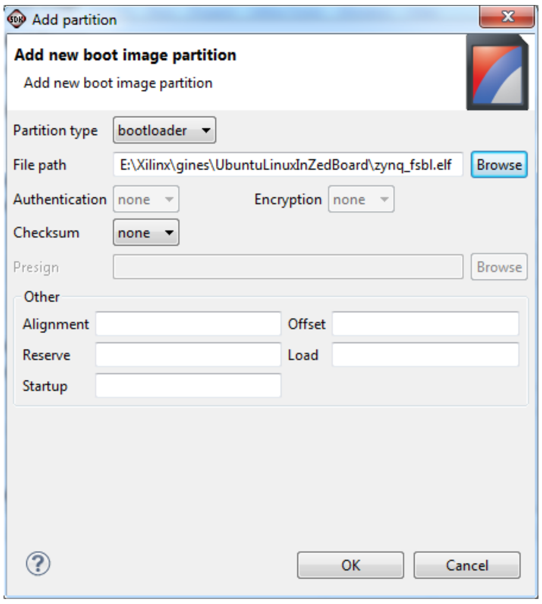


Figure 174: Create Zynq Boot Image Selection

For adding the Boot image partitions, dick in "Add" three times, adding each file as showed in the following figures. It is important to remember that the order of the images should always be the same.

1. First Stage Boot Loader ("zynq_fsbl.elf").
2. The Programmable Logic BitStream ("system.bit").
3. The software application file, in this case, the Second Stage Boot Loader U-Boot ("u-boot.elf").

Fill in the information such as the next figures, by dicking in "Add":


Add partition

Add new boot image partition

Add new boot image partition

Partition type: bootloader

File path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\zynq_fsbl.elf

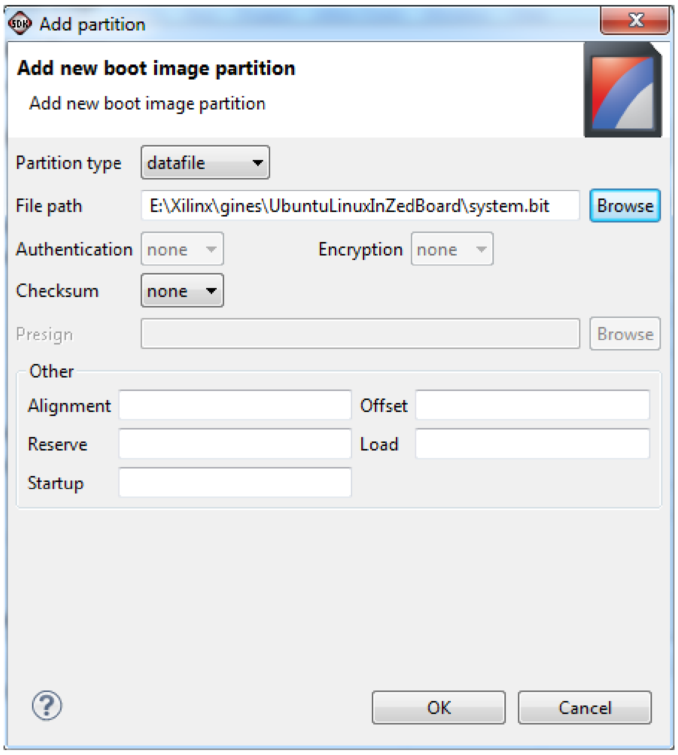
Authentication: none — Encryption: none

Checksum: none

Presign:

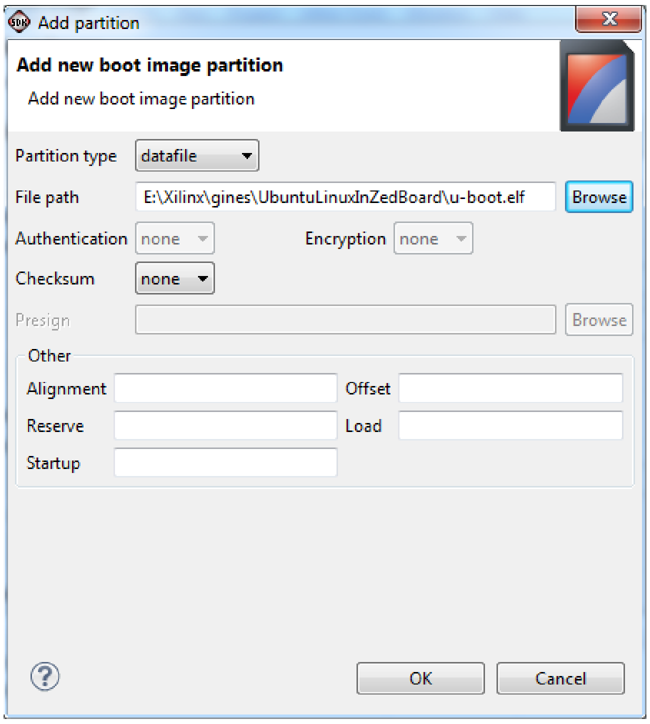
Other — Alignment: , Offset: , Reserve: , Load: , Startup:

Figure 175: Adding the FSBL



Partition type: datafile
File path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\system.bit
Authentication: none
Encryption: none
Checksum: none
Presign:
Alignment:
Offset:
Reserve:
Load:
Startup:

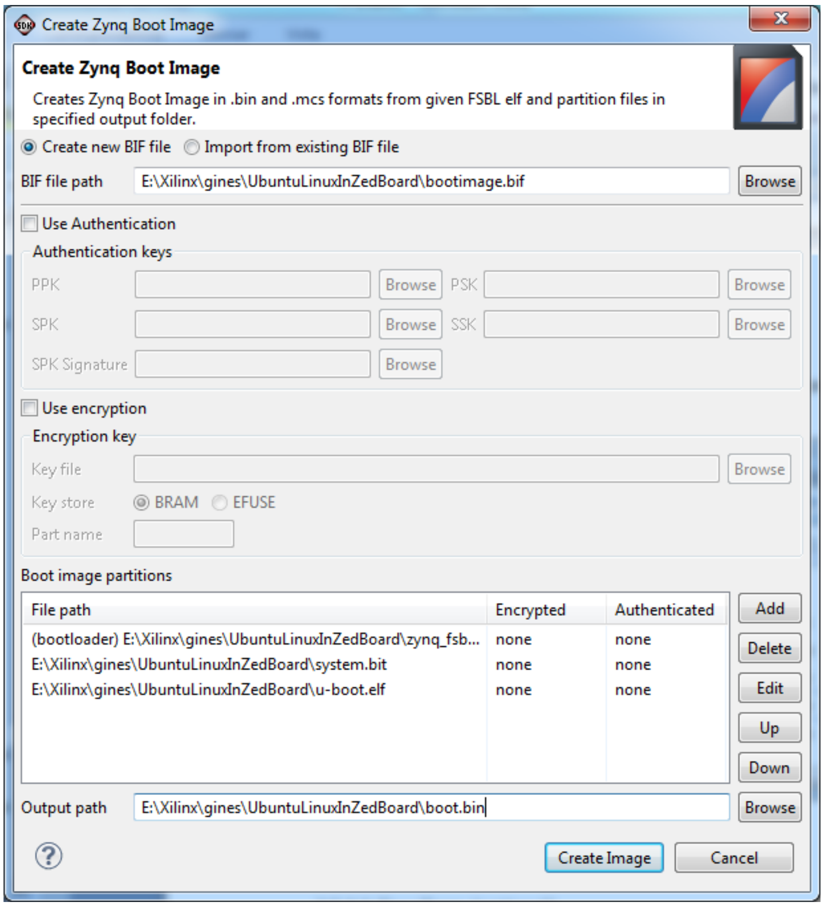
Figure 176: Adding the BitStream File



Partition type: datafile
File path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\u-boot.elf
Authentication: none
Encryption: none
Checksum: none
Presign:
Alignment:
Offset:
Reserve:
Load:
Startup:

Figure 177: Adding the U-Boot File

Finally, the final appearance will be the following:



Create Zynq Boot Image

Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

[✓ Create new BIF file] Import from existing BIF file

BIF file path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\bootimage.bif

Use Authentication (unchecked)

Use encryption (unchecked)

Key store: [✓ BRAM] EFUSE

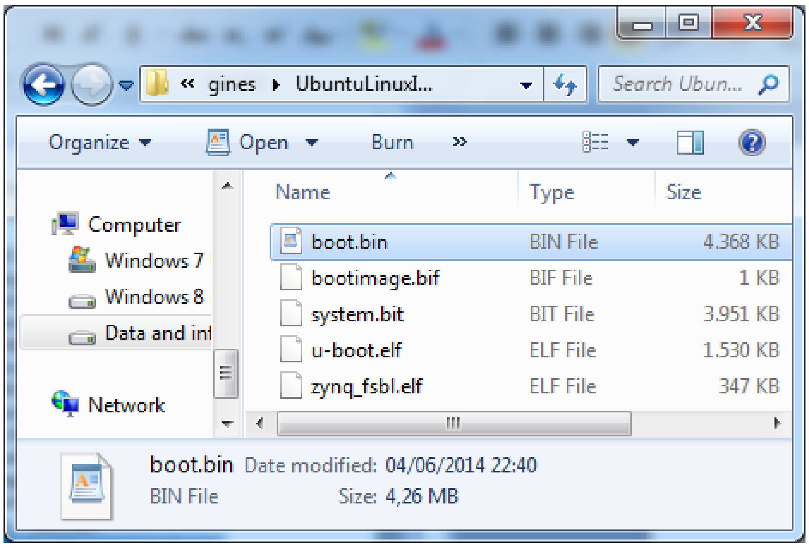
Boot image partitions

File path	Encrypted	Authenticated
(bootloader) E:\Xilinx\gines\UbuntuLinuxInZedBoard\zynq_fsb...	none	none
E:\Xilinx\gines\UbuntuLinuxInZedBoard\system.bit	none	none
E:\Xilinx\gines\UbuntuLinuxInZedBoard\u-boot.elf	none	none

Output path: E:\Xilinx\gines\UbuntuLinuxInZedBoard\boot.bin

Figure 178: Create the Zynq Boot Image

Finally, click in “Create Image”. Once SDK has been concluded, the file “boot.bin” will be found in the same folder.



Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.530 KB
zynq_fsbl.elf	ELF File	347 KB

boot.bin Date modified: 04/06/2014 22:40
BIN File Size: 4,26 MB

Figure 179: “boot.bin” Folder

2.6.3.2. Tera Term (optional)

“boot.bin” can also be copied into the SD Card memory and tested in the board like in the previous section. Notice that the SD Card has to be previously formatted in FAT32 file system. For tested the boot file, “Tera Term” can be used. Once Tera Term is launched, switch on the board and click in “File”, “New conection…”, “Serial” and “OK”.

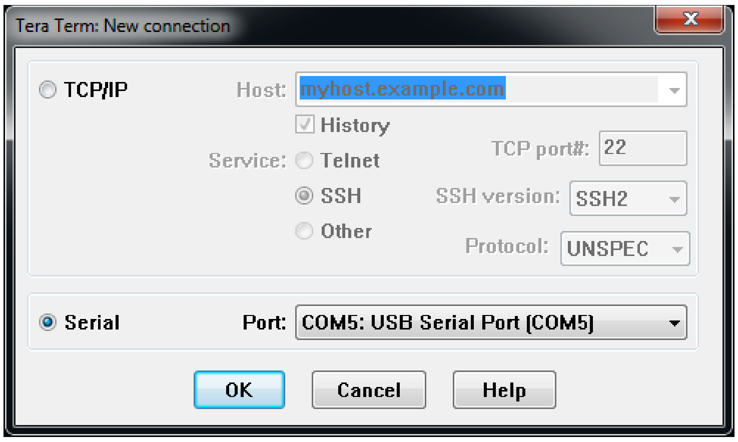
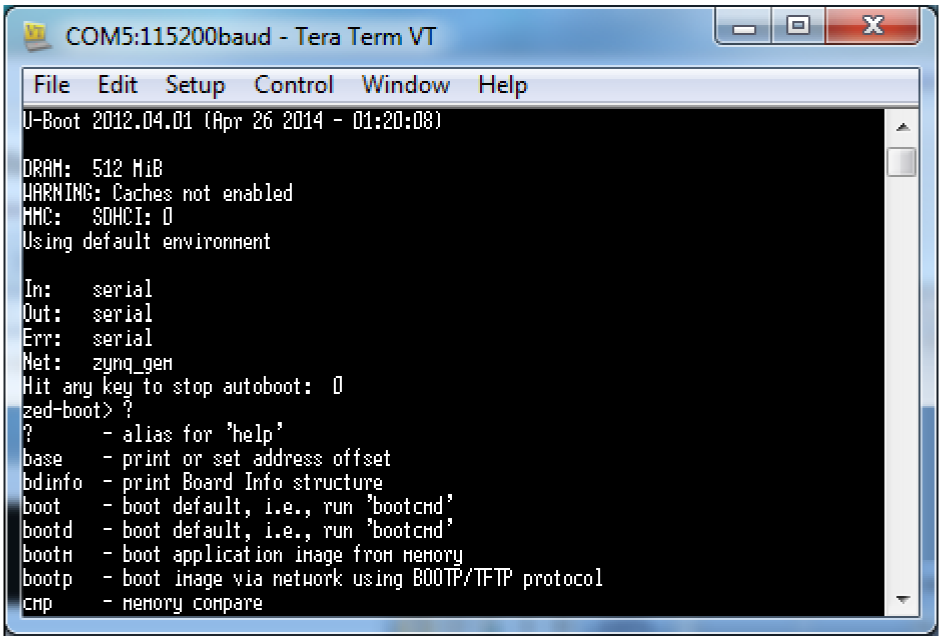


Figure 180: Tera Term - New Connection

Now, click in the reset button of the ZedBoard or switch it off and switch it on again. If no button is pushed, the boot loader will try to load the OS but errors will be displayed because there is no OS inside the SD Card. Nevertheless, whether a button in the keyboard is pushed, the boot loaded will not load the operating system and the user will be able to use the command plot. Writing “?” in the command plot, the command list will be showed:


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
U-Boot 2012.04.01 (Apr 26 2014 - 01:20:08)

DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   SDHCI: 0
Using default environment

In:    serial
Out:   serial
Err:   serial
Net:   zynq_gem
Hit any key to stop autoboot:  0
zed-boot> ?
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
```

Figure 181: Testing the “boot.bin” File

Therefore, the boot file works properly.

3.7. Device Tree Binary, "devicetree.dtb"

3.7.1. VMware Player

The device tree compiler (dtc), located under scripts/dtc in the Linux kernel source, is responsible for carrying out this work as it occurred with the previous Linux OS. The commands are the followings:

```
cd ~/ubuntu/

git checkout xcomm_zynq

make ARCH=arm distclean

make ARCH=arm zynq_xcomm_adv7511_defconfig

make ARCH=arm zynq-zed-adv7511.dtb
```

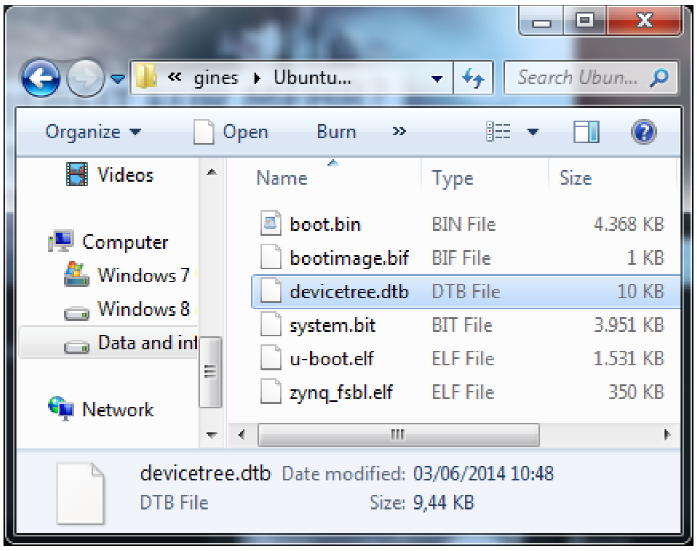
Command Window 17: Building the DTB File from the Default DTS File

- **"cd ~/ubuntu/"** selects the Ubuntu directory.
- **"git checkout xcomm_zynq"** sets up the xcomm_zynq branch for remote tracking, importing the required files that this configuration requires.
- **"make ARCH=arm distclean"** runs a make distribution clean command against the kernel source code for good measure. This command will remove all intermediary files created by setting as well as any intermediary files created by make and it is a good way to clean up any stale configurations.
- **"make ARCH=arm zynq_xcomm_adv7511_defconfig"** configures the kernel for the ZedBoard. The command prepares the kernel source tree for the Zynq-7000 architecture including some special configuration for the ZedBoard. It builds a default configuration ".config" file. The architecture type and the cross compiler prefix are also specified respectively with "ARCH=arm" and "CROSS_COMPILER=arm-linux-".
- **"make ARCH=arm zynq-zed-adv7511.dtb"** order to the dtc compiles to convert the selected default DTS file into a DTB file.

This command will search the default "zynq-zed-adv7511.dts" file in the previous indicated path, ubuntu/arch/arm/boot/dts/, and it will generate the "zynq-zed-adv7511.dtb" file in the next path:

.../ubuntu/arch/arm/dts/zynq-zed-adv7511.dtb

Once the DTB file is generated with any of these two ways, copy it, paste into the Windows folder "UbuntuLinuxInZedBoard" and rename it to "devicetree.dtb".



Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
devicetree.dtb	DTB File	10 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
zynq_fsbl.elf	ELF File	350 KB

devicetree.dtb Date modified: 03/06/2014 10:48
DTB File Size: 9,44 KB

Figure 182: Device Tree Binary File in the "UbuntuLinuxInZedBoard" Folder

3.8. Linux Kernel File, "uImage"

3.8.1. VMware

Once again, the method is the same than the previously used in the previous Linux OS. Therefore, the required commands are showed below:

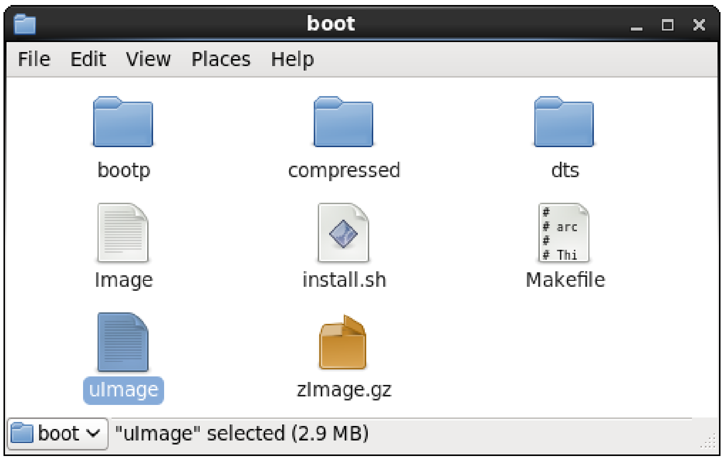
```
      cd ~/ubuntu/
      git checkout xcomm_zynq
      make ARCH=arm distclean
      make ARCH=arm zynq_xcomm_adv7511_defconfig
      make ARCH=arm
      cd arch/arm/boot
      gzip zImage
      ~/u-boot-xlnx/tools/mkimage -A arm -a 0x8000 -e 0x8000 -n
'Linux kernel' -T kernel -d ~/ubuntu/arch/arm/boot/zImage.gz
~/ubuntu/arch/arm/boot/uImage
      make ARCH=arm zynq-zed-adv7511.dtb
```

Command Window 18: Building the "uImage" File

- **“cd ~/ubuntu/”** selects the Ubuntu directory.
- **“git checkout xcomm_zynq”** sets up the xcomm_zynq branch for remote tracking, importing the required files that this configuration requires.
- **“make ARCH=arm distclean”** runs a make distribution clean command against the kernel source code for good measure. This command will remove all intermediary files created by setting as well as any intermediary files created by make and it is a good way to clean up any stale configurations.
- **“make ARCH=arm zynq_xcomm_adv7511_defconfig”** configures the kernel for the ZedBoard. The command prepares the kernel source tree for the Zynq-7000 architecture including some special configuration for the ZedBoard. It builds a default configuration “.config” file. The architecture type and the cross compiler prefix are also specified respectively with “ARCH=arm”.
- **“make ARCH=arm”** generates the desired Linux Kernel file for the Zynq ARM architecture, once the environment variables are properly set.
- **“cd arch/arm/boot”** selects the indicated directory.
- **“gzip zImage”** re-compresses the “zImage” file into a “zImage.gz” file.
- **“~/u-boot-xlnx/tools/mkimage -A arm -a 0x8000 -e 0x8000 -n 'Linux kernel' -T kernel -d ~/ubuntu/arch/arm/boot/zImage.gz ~/ubuntu/arch/arm/boot/ uImage”** wraps the “zImage.gz” filo into a U-Boot-wrapped “uImage”. The mkimage tool is used from the U-Boot repository.

Once the Kernel has been built, it will be showed in the following path directory.

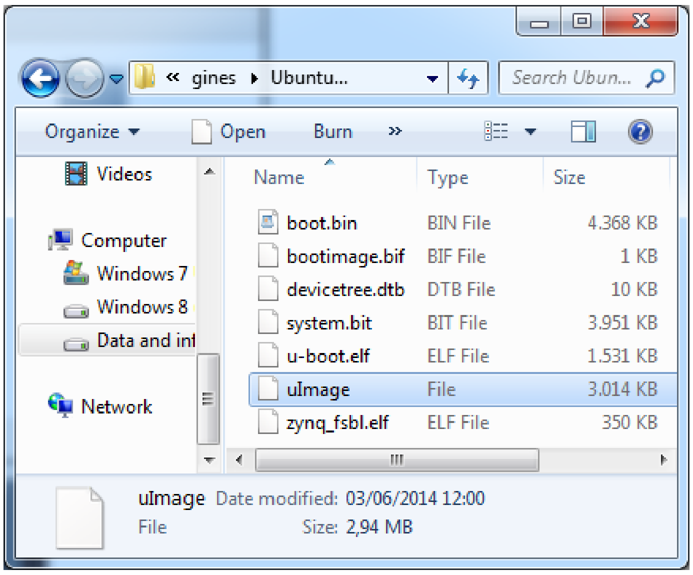
.../linux-xilinx/arch/arm/boot/uImage



"uImage" selected (2.9 MB)

Figure 183: “uImage” Location Folder

Once again, it must be copied into the “UbuntuLinuxInZedBoard” folder.



Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
devicetree.dtb	DTB File	10 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
uImage	File	3.014 KB
zynq_fsbl.elf	ELF File	350 KB

uImage Date modified: 03/06/2014 12:00
File Size: 2,94 MB

Figure 184: "uImage" in the "UbuntuLinuxInZedBoard" folder

3.9. Root File System Image

3.9.1. VMware Player

This section is the one which changes most from the previous Linux OS, because it will be directly obtained from the Linaro website. The commands for downloaded and extract into the SD Card this Ubuntu version are showed below. However, after executing them, the SD Card must be re-connected to the virtual machine by right-clicking in "Connect (Disconnect from Host)", like it was executed in the previous subchapter "3.2. Preparing the SD Card".

```
      cd ~

      wget http://releases.linaro.org/12.09/ubuntu/precise-
images/ubuntu-desktop/linaro-precise-ubuntu-desktop-20120923-
436.tar.gz

      sudo tar --strip-components=3 -C /media/rootfs -xzpf linaro-
precise-ubuntu-desktop-20120923-436.tar.gz
binary/boot/filesystem.dir
```

Command Window 19: Linaro-Ubuntu 12.09 Release

Anyway, other Linaro-Ubuntu Desktop editions can be also implemented:

```
      cd ~

      wget http://releases.linaro.org/11.12/ubuntu/oneiric-
images/ubuntu-desktop/linaro-o-ubuntu-desktop-tar-20111219-0.tar.gz

      sudo tar --strip-components=3 -C /media/rootfs -xzpf linaro-o-
ubuntu-desktop-tar-20111219-0.tar.gz
```

Command Window 20: Linaro-Ubuntu 11.12 Release

```
      cd ~

      http://releases.linaro.org/12.03/ubuntu/oneiric-images/ubuntu-
desktop/linaro-o-ubuntu-desktop-tar-20120327-0.tar.gz

      sudo tar --strip-components=3 -C /media/rootfs -xzpf linaro-o-
ubuntu-desktop-tar-20120327-0.tar.gz
```

Command Window 21: Linaro-Ubuntu 12.03 Release

```
      cd ~

      wget http://releases.linaro.org/12.11/ubuntu/precise-
images/ubuntu-desktop/linaro-precise-ubuntu-desktop-20121124-
560.tar.gz

      sudo tar --strip-components=3 -C /media/rootfs -xzpf linaro-
precise-ubuntu-desktop-20121124-560.tar.gz
binary/boot/filesystem.dir
```

Command Window 22: Linaro-Ubuntu 12.11 Release

- **“cd ~”** simply changes the directory to the main folder.
- **“wget” + URL** downloads the indicated file, an Ubuntu Linux OS version.
- **“sudo tar --strip-components=3 -C /media/rootfs” + file_path** decompresses and extracts into the “rootfs” folder (one of the SD Card partitions) the previous file.

Note that these commands will spend some minutes to be completed because of the size of the Operating System.

Sometimes, the third command provokes an error, indicating that the ”/media/rootfs/” directory does not exist. This error can be solved by opening manually the rootfs folder from the “Computer” folder and re-executing the command in the previous “Terminal” window.

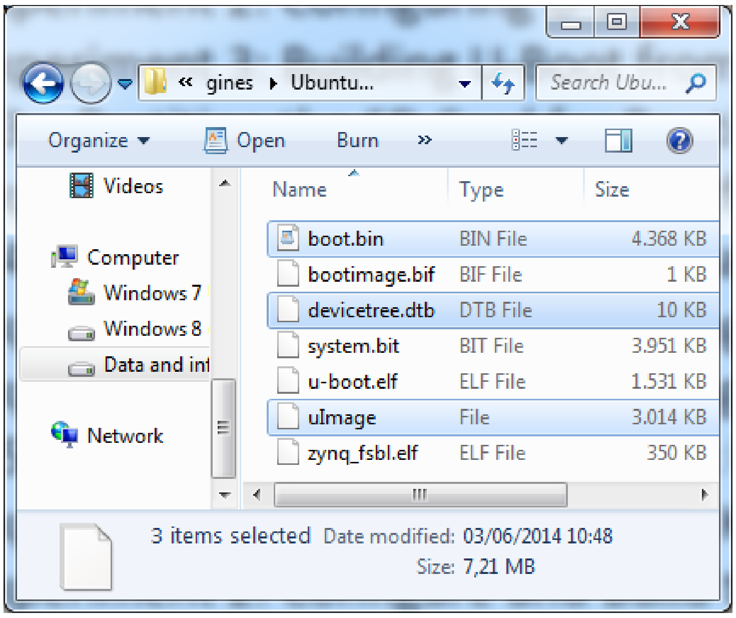
Finally, the SD Card can be disconnected from the virtual machine by clicking with the right mouse button in “Disconnect (Connect to host)” in the SD Card icon. Remember that this icon is in the highest and rightmost position.

3.10. Booting Ubuntu on ZedBoard

The Root File System Image has been copied into the “rootfs” partition of the SD Card. Now, the “UbuntuLinuxInZedBoard” must be opened in Windows and the other required files must be copied into the “BOOT” partition of the SD Card, which can be opened from Windows. Remember that the required files are the following:

- The FSBL, together with the BitStream and U-Boot files, “boot.bin” file.
- The Device Tree Binary file, “devicetree.dtb” file.

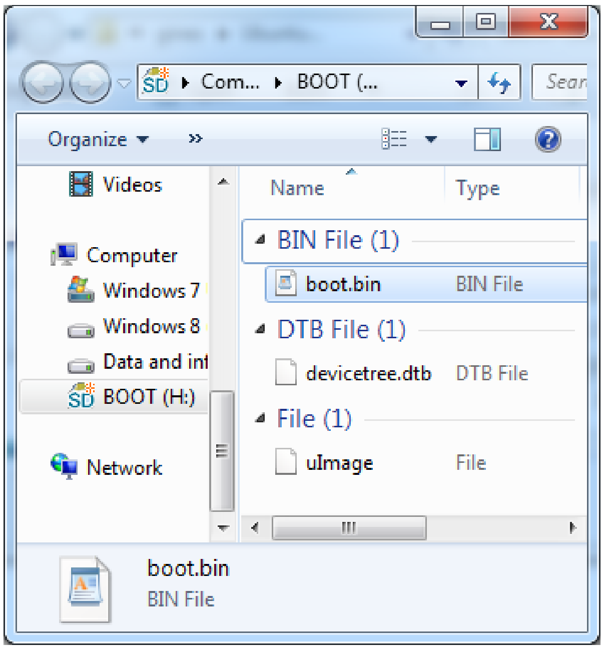
- The Linux Kernel file, “uImage” file.



Name	Type	Size
boot.bin	BIN File	4.368 KB
bootimage.bif	BIF File	1 KB
devicetree.dtb	DTB File	10 KB
system.bit	BIT File	3.951 KB
u-boot.elf	ELF File	1.531 KB
uImage	File	3.014 KB
zynq_fsbl.elf	ELF File	350 KB

3 items selected Date modified: 03/06/2014 10:48
Size: 7,21 MB

Figure 185: Required Files in the SD Card



Name	Type
BIN File (1)	
boot.bin	BIN File
DTB File (1)	
devicetree.dtb	DTB File
File (1)	
uImage	File

boot.bin
BIN File

Figure 186: Required Files Copied in the SD Card

3.10.1. Tera Term

The SD Card already contains the required files. Thus, it can be inserted into the ZedBoard. After that, the board can be connected to the Tera Term program, to a HDMI display, a USB mouse and a USB keyboard; and switched on.

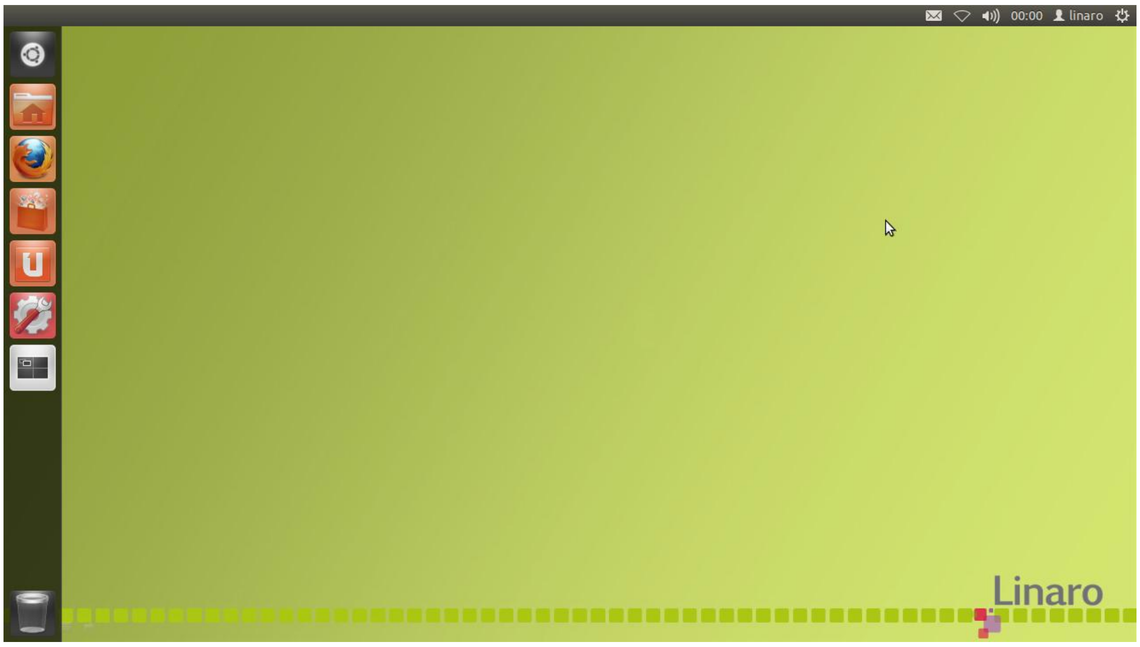


Figure 187: Linaro-Ubuntu 12.09 Appearance

Now, ZedBoard can be used as a normal Laptop with a Desktop Operating System.

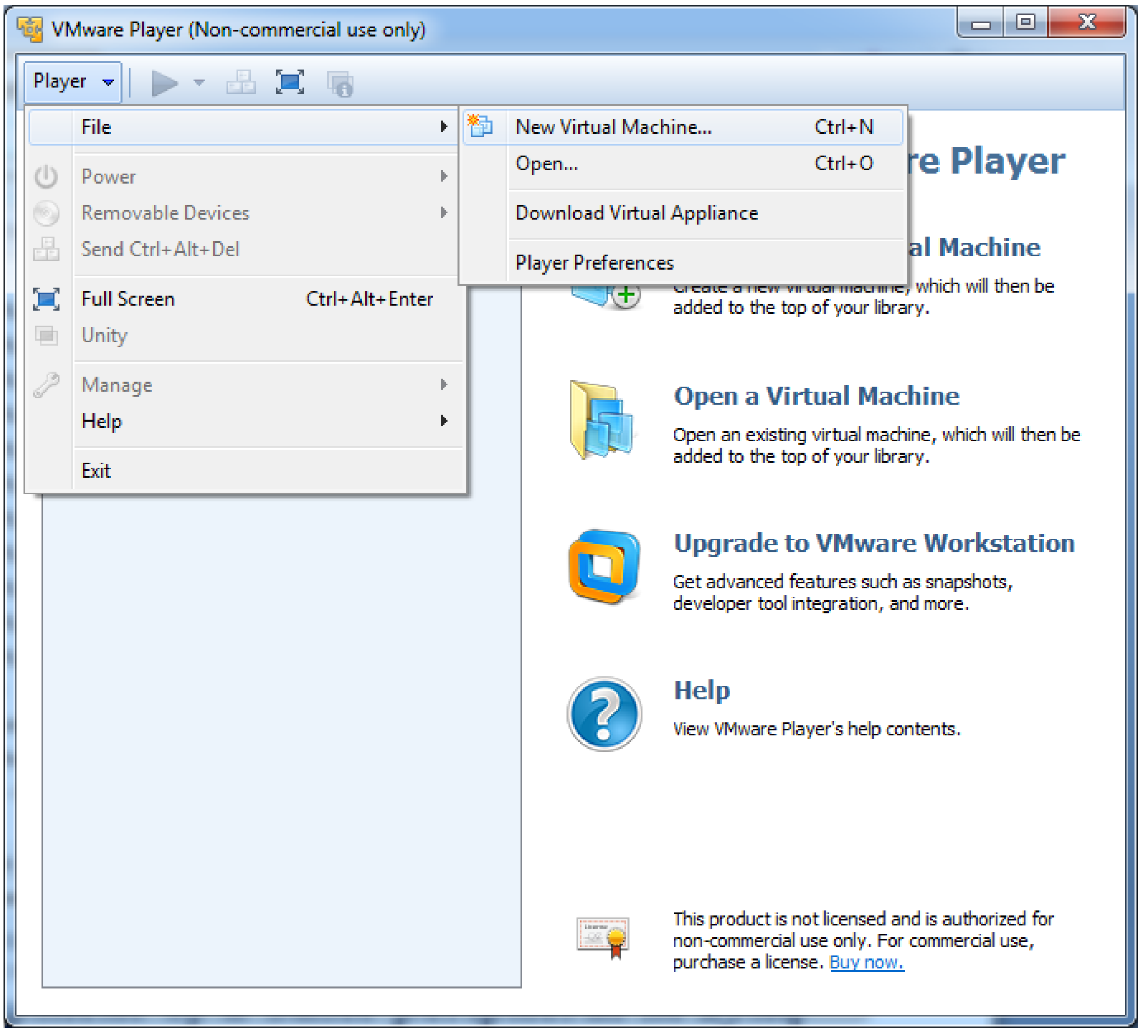
Appendix 2: Prerequisites

A2.1. VMware Player

A2.1. Installing VMware Player

VMware player can be downloaded for free from its website: http://www.vmware.com. Meanwhile, the CentOS Linux can be also downloaded for free in http://www.centos.org/.

Once VMware is installed, open it and click in “Player”, “File” and “New Virtual Machine”:


VMware Player (Non-commercial use only)
Player
File
Power
Removable Devices
Send Ctrl+Alt+Del
Full Screen Ctrl+Alt+Enter
Unity
Manage
Help
Exit
New Virtual Machine... Ctrl+N
Open... Ctrl+O
Download Virtual Appliance
Player Preferences
re Player
al Machine

Create a new virtual machine, which will then be added to the top of your library.

Open a Virtual Machine

Open an existing virtual machine, which will then be added to the top of your library.

Upgrade to VMware Workstation

Get advanced features such as snapshots, developer tool integration, and more.

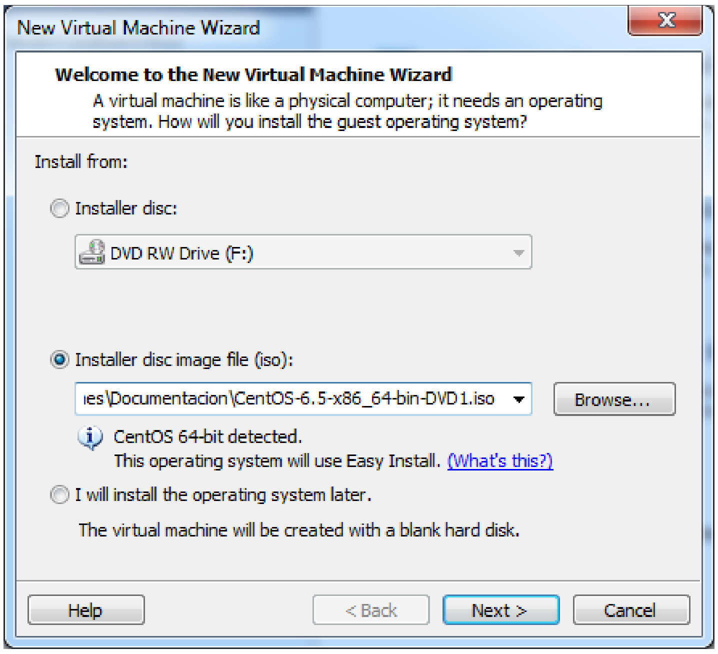
Help

View VMware Player's help contents.

This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

Figure 188: New Virtual Machine Selection

In “New Virtual Machine Wizard” select “Installer disc image file (iso)” and locate the CentOS installation media image file “CentOS-6.5-x86_64-bin-DVD1.iso”:


New Virtual Machine Wizard

Welcome to the New Virtual Machine Wizard

A virtual machine is like a physical computer; it needs an operating system. How will you install the guest operating system?

Install from:

( ) Installer disc:

DVD RW Drive (F:)

(●) Installer disc image file (iso):

ıes\Documentacion\CentOS-6.5-x86_64-bin-DVD1.iso

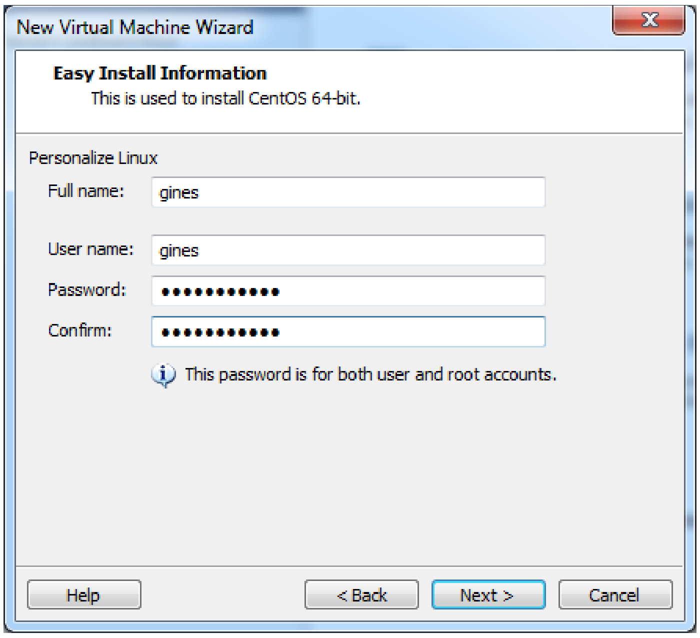
CentOS 64-bit detected.
This operating system will use Easy Install. (What's this?)

( ) I will install the operating system later.

The virtual machine will be created with a blank hard disk.

Figure 189: New Virtual Machine Wizard Window

Under “Easy Install”, some steps are skipped. A name and password must be introduced, for instance, “gines” as user (it will be easier if “Full name” and “User name” are the same):


New Virtual Machine Wizard

Easy Install Information

This is used to install CentOS 64-bit.

Personalize Linux

Full name: gines

User name: gines

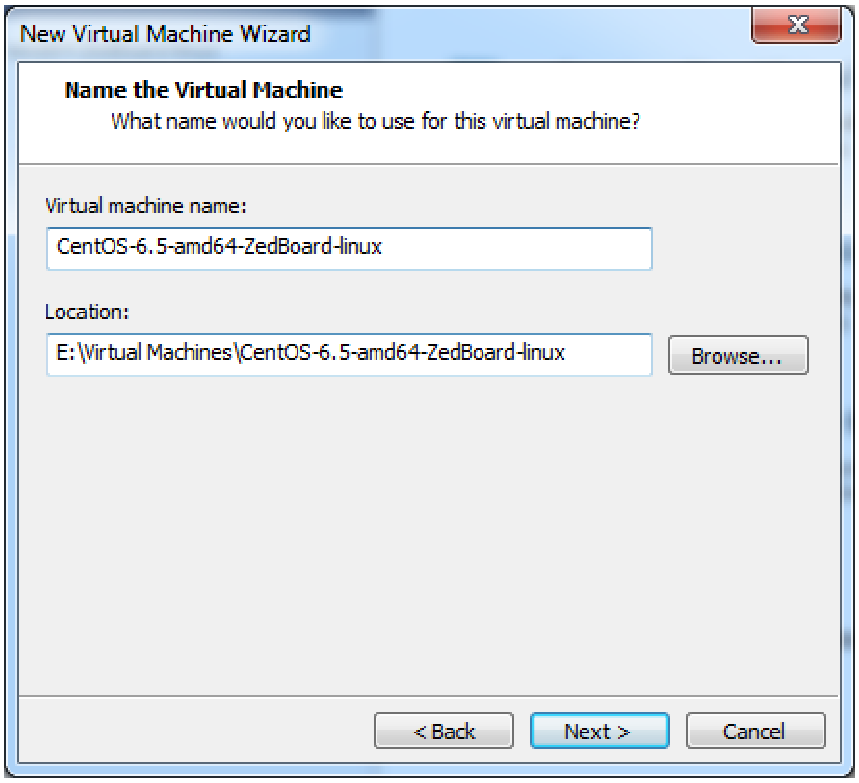
Password: ●●●●●●●●●●●

Confirm: ●●●●●●●●●●●

This password is for both user and root accounts.

Figure 190: New Virtual Machine – User Name Selection

A name for the virtual machine and its location have to be selected:


New Virtual Machine Wizard

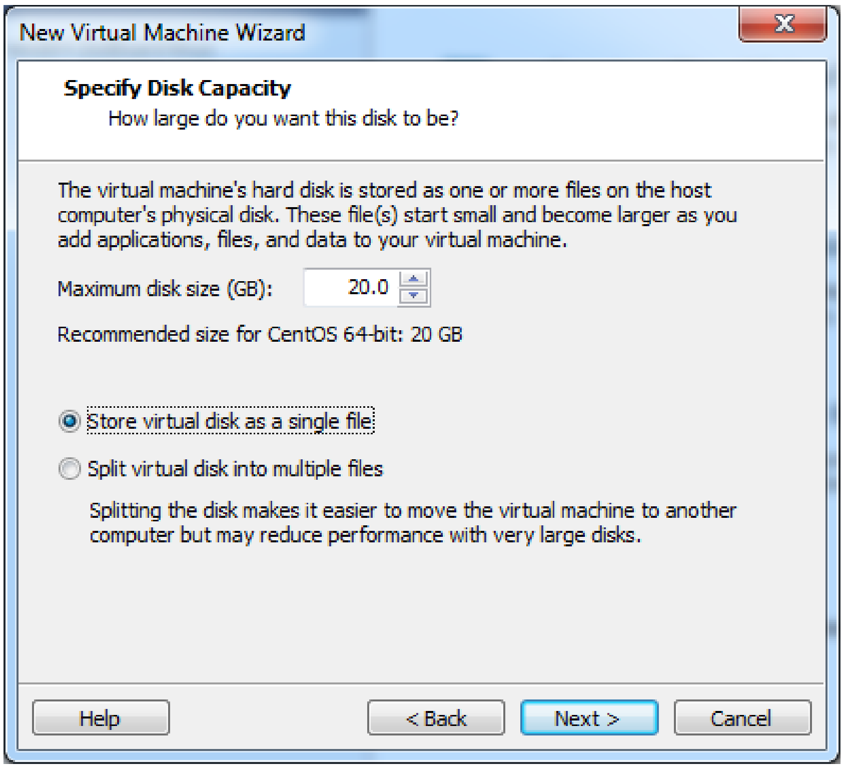
Name the Virtual Machine
What name would you like to use for this virtual machine?

Virtual machine name: CentOS-6.5-amd64-ZedBoard-linux

Location: E:\Virtual Machines\CentOS-6.5-amd64-ZedBoard-linux

Figure 191: New Virtual Machine – VM Name Selection

At least 20.0GB and "Store virtual disk as single file" must be fixed:


New Virtual Machine Wizard

Specify Disk Capacity
How large do you want this disk to be?

The virtual machine's hard disk is stored as one or more files on the host computer's physical disk. These file(s) start small and become larger as you add applications, files, and data to your virtual machine.

Maximum disk size (GB): 20.0

Recommended size for CentOS 64-bit: 20 GB

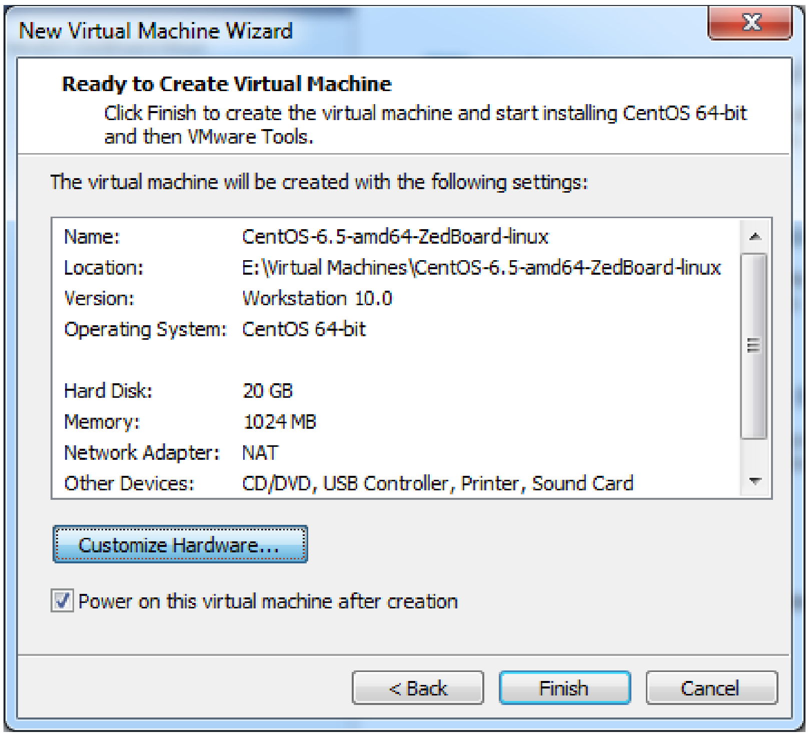
[✓ Store virtual disk as a single file]

Split virtual disk into multiple files

Splitting the disk makes it easier to move the virtual machine to another computer but may reduce performance with very large disks.

Figure 192: New Virtual Machine - Hard Disk Selection

In the Ready to Create Virtual Machine window, configure the settings for the machine by clicking "Customize Hardware":


New Virtual Machine Wizard

Ready to Create Virtual Machine

Click Finish to create the virtual machine and start installing CentOS 64-bit and then VMware Tools.

The virtual machine will be created with the following settings:

Name:	CentOS-6.5-amd64-ZedBoard-linux
Location:	E:\Virtual Machines\CentOS-6.5-amd64-ZedBoard-linux
Version:	Workstation 10.0
Operating System:	CentOS 64-bit
Hard Disk:	20 GB
Memory:	1024 MB
Network Adapter:	NAT
Other Devices:	CD/DVD, USB Controller, Printer, Sound Card

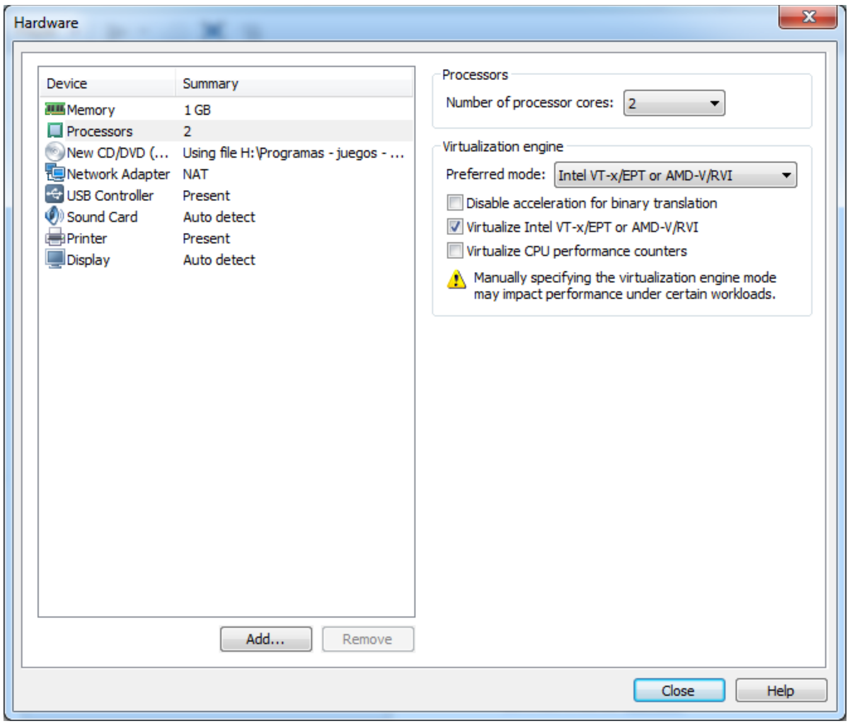
Customize Hardware...

[✓] Power on this virtual machine after creation

< Back | Finish | Cancel

Figure 193: New Virtual Machine – Settings Displayed

At least 1024MB for the memory must be chosen. Additionally, in "Processors", "Preferred mode" must be "Intel VT-x/EPT or AMD-V/RVI" and click in "Virtualize Intel VT-X/EPT or AMD-V/RVI":


Hardware

Device	Summary
Memory	1 GB
Processors	2
New CD/DVD (...	Using file H:\Programas - juegos - ...
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

Add... | Remove

Processors

Number of processor cores: 2

Virtualization engine

Preferred mode: Intel VT-x/EPT or AMD-V/RVI

[ ] Disable acceleration for binary translation

[✓] Virtualize Intel VT-x/EPT or AMD-V/RVI

[ ] Virtualize CPU performance counters

Manually specifying the virtualization engine mode may impact performance under certain workloads.

Close | Help

Figure 194: New Virtual Machine – Customize Hardware Selection

The virtual machine will be created and CentOS installed in the new machine. This process will take about 30 minutes to complete depending upon the host machine performance.

A2.2. Configuring VMware Player

Once the installation process is complete, the virtual machine will be powered on and it will look like the following figure after the login:

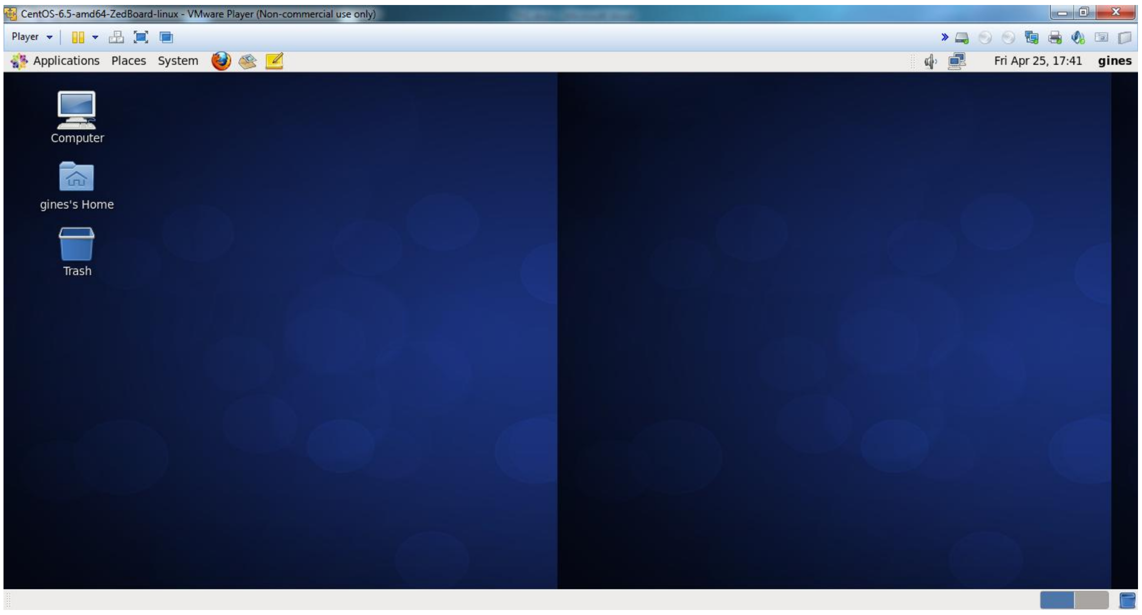


Figure 195: Virtual Machine Running

Open a terminal window by clicking “Applications”, “System Tools” and “Terminal”:

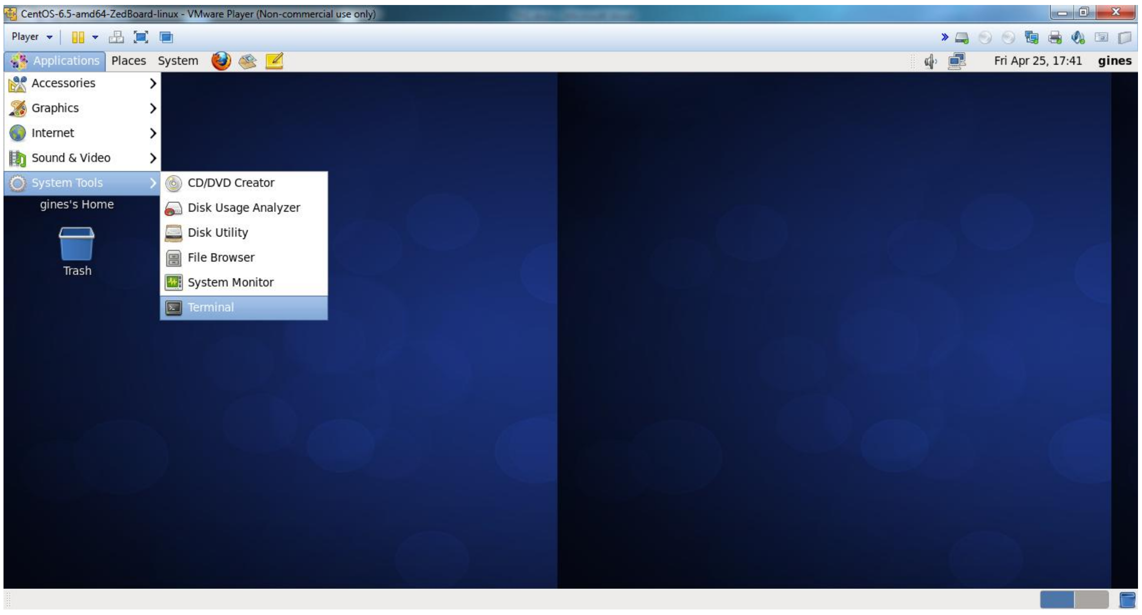


Figure 196: Opening the “Terminal” Program

Take on root privileges by running the super-user elevation with the following command:

```
su
```

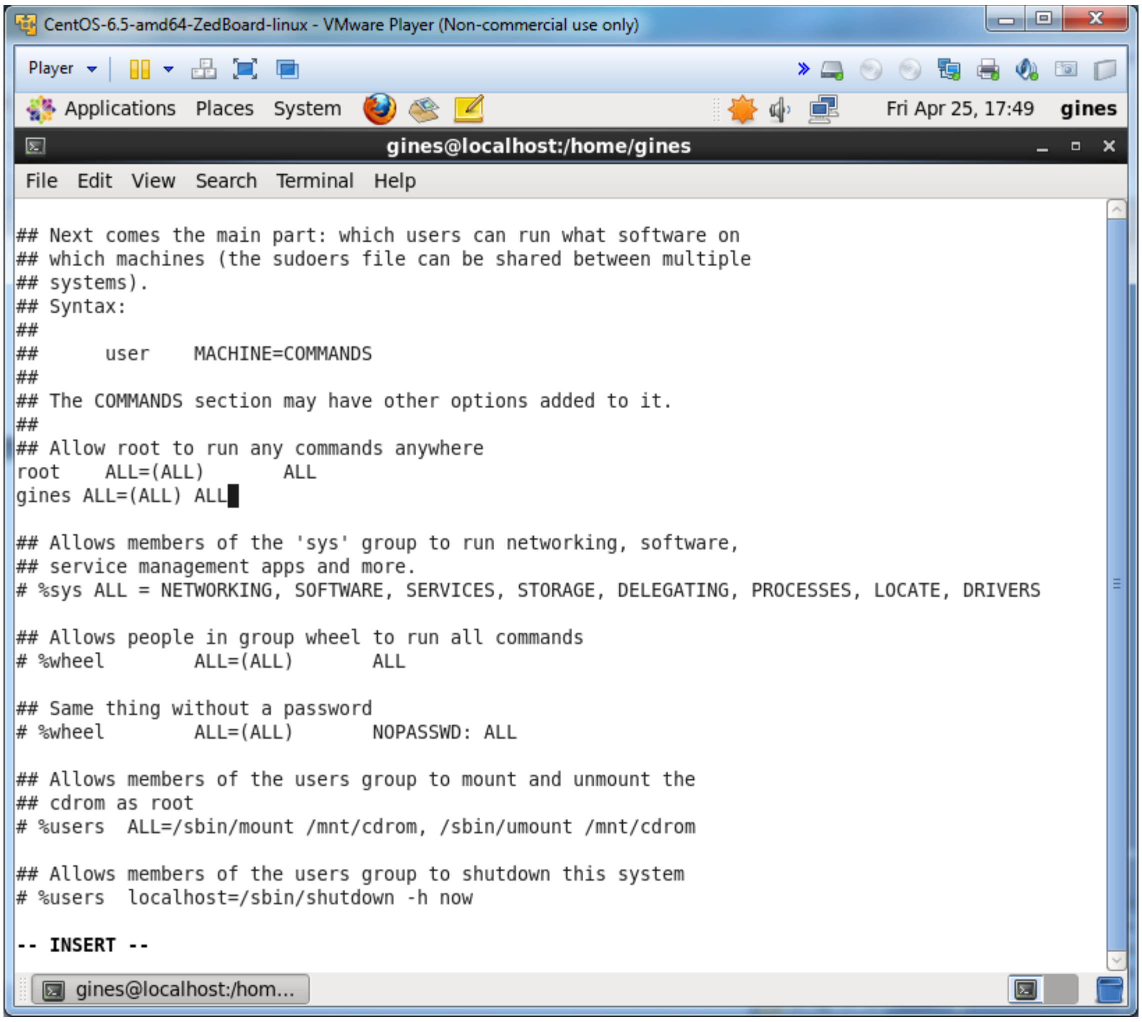
Command Window 23: Giving Root Privileges

Use the visudo text editor to edit the /etc/sudoers file:

```
visudo
```

Command Window 24: Edit the "sudoers" File

Add the "gines" user to the sudoers list by inserting the line: "gines ALL=(ALL) ALL" to the users section as shown in the next figure (to insert text in the editor, press the "I" key on the keyboard):


CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Player
Applications Places System
Fri Apr 25, 17:49 gines
gines@localhost:/home/gines
File Edit View Search Terminal Help
gines@localhost:/hom...

```
## Next comes the main part: which users can run what software on
## which machines (the sudoers file can be shared between multiple
## systems).
## Syntax:
##
##      user    MACHINE=COMMANDS
##
## The COMMANDS section may have other options added to it.
##
## Allow root to run any commands anywhere
root    ALL=(ALL)       ALL
gines ALL=(ALL) ALL

## Allows members of the 'sys' group to run networking, software,
## service management apps and more.
# %sys ALL = NETWORKING, SOFTWARE, SERVICES, STORAGE, DELEGATING, PROCESSES, LOCATE, DRIVERS

## Allows people in group wheel to run all commands
# %wheel        ALL=(ALL)       ALL

## Same thing without a password
# %wheel        ALL=(ALL)       NOPASSWD: ALL

## Allows members of the users group to mount and unmount the
## cdrom as root
# %users  ALL=/sbin/mount /mnt/cdrom, /sbin/umount /mnt/cdrom

## Allows members of the users group to shutdown this system
# %users  localhost=/sbin/shutdown -h now

-- INSERT --
```

Figure 197: Adding an User

And close it by pulsing the key “Esc”, writing “Shift” + “.wq”, writing “exit” and pulsing “Enter”.

A2.3. Required Git Repositories

A2.3.1. Sourcery CodeBench Lite for Xilinx GNU Linux

Next, the system will be updated to the latest updates so use the yum package manager to install the system updates. These updates can take several minutes and may present several user prompts before completion. If prompted to allow download of packages and/or for the import of the GPG key, accept by pressing “Y” followed by the Enter key.

```
sudo yum update
```

Command Window 25: Updating Sourcery CodeBench

The “ncurses-devel” package and Git SCM tool will be installed next using the package manager and when prompted accept the download and installation of all recommended packages.

```
sudo yum install ncurses-devel git
```

Command Window 26: Installing “ncurses-devel”

For using this as the own development machine to submit patches, Git can be configured with the name and email address. In this case:

```
git config --global user.name ‘gines’
git config --global user.email ‘gineshidalgo99@gmail.com’
```

Command Window 27: Setting the Name and E-mail

The editor preference can be set (default is vi or vim) if desired. On this reference system, the vi editor will be used.

```
git config --global core.editor vi
```

Command Window 28: Setting the Editor

If there is any preference for any particular diff tool used to resolve merge conflicts, this should also be set. On this reference system, the vimdiff tool will be used.

```
git config --global merge.tool vimdiff
```

Command Window 29: Setting the Diff Tool

For using Sourcery CodeBench on an x86 64-bit Linux host system, the 32-bit system libraries must be installed. They are available as a series of packages which can be installed using yum. When prompted, accept the defaults to install all packages.

```
sudo yum install glibc-devel.i686 gtk2-devel.i686 \
gtk-nodoka-engine.i686 libcanberra.i686 \
libcanberra-gtk2.i686 PackageKit-gtk-module.i686 \
GConf2.i686 ncurses-libs.i686 xulrunner.i686
```

Command Window 30: Installing the 32-Bit System Library

The Sourcery CodeBench cross toolchain installer "xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin" must be obtained from the Xilinx URL below:

http://www.xilinx.com/member/mentor_codebench/xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin

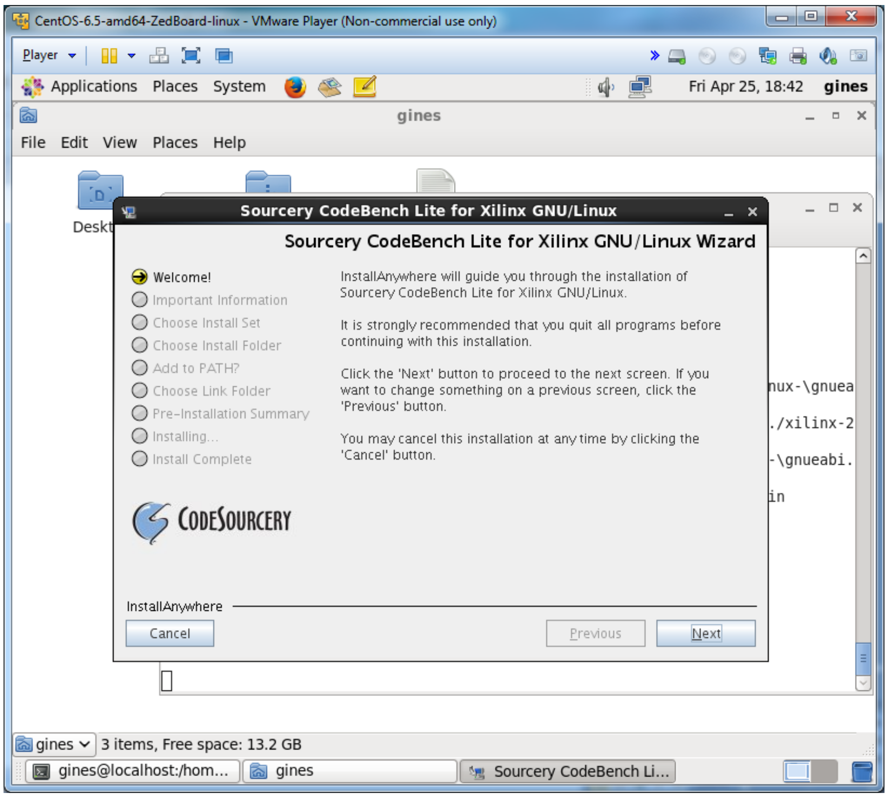
It can be also directly downloaded in the next link (this link does not require account login):

https://code.google.com/p/zedboard-book-source/downloads/detail?name=xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin&can=2&q=

Furthermore, the file can be downloaded from Windows host operating system and directly copy and paste inside the virtual machine or can be directly downloaded in the Virtual Machine. Once downloaded, launch the CodeSourcery cross toolchain installer:

```
mv /home/gines/xilinx-2011.09-50-arm-xilinx-linux-\gnueabi.bin .
chmod ugo+x xilinx-2011.09-50-arm-xilinx-linux-\gnueabi.bin
./xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin
```

Command Window 31: Launching the CodeSourcery Cross Toolchain Installer


CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Applications Places System
Fri Apr 25, 18:42 gines
gines
File Edit View Places Help
Sourcery CodeBench Lite for Xilinx GNU/Linux

Sourcery CodeBench Lite for Xilinx GNU/Linux Wizard

- Welcome!
- Important Information
- Choose Install Set
- Choose Install Folder
- Add to PATH?
- Choose Link Folder
- Pre-Installation Summary
- Installing...
- Install Complete

InstallAnywhere will guide you through the installation of Sourcery CodeBench Lite for Xilinx GNU/Linux.

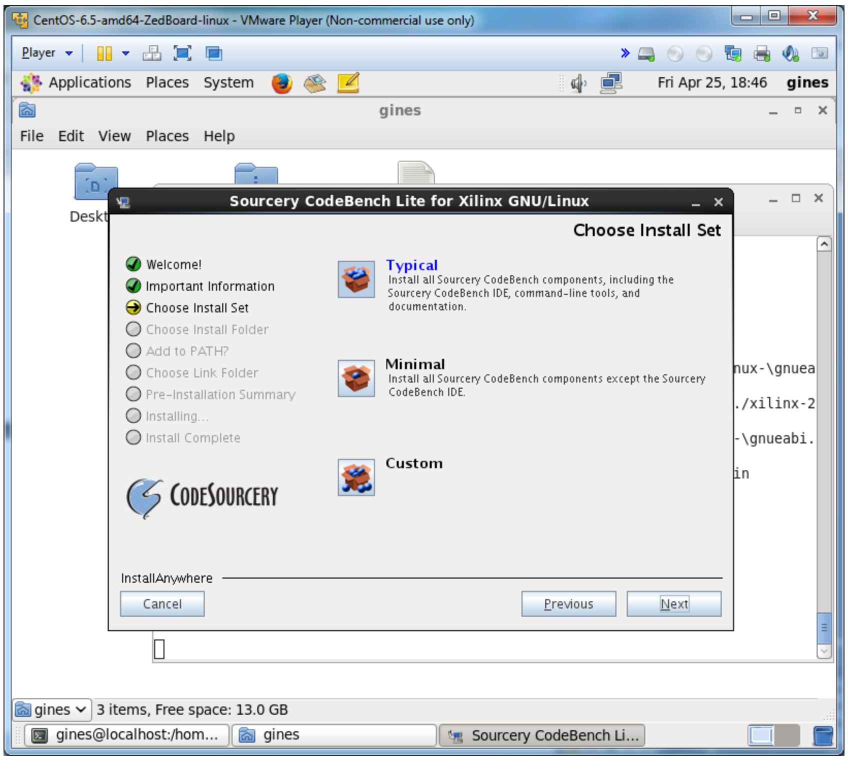
It is strongly recommended that you quit all programs before continuing with this installation.

Click the 'Next' button to proceed to the next screen. If you want to change something on a previous screen, click the 'Previous' button.

You may cancel this installation at any time by clicking the 'Cancel' button.

Figure 198: Sourcery CodeBench Wizard

Click in “Next” several times and “Install”, verifying the “Typical” install:


CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Applications Places System
Fri Apr 25, 18:46 gines
gines
File Edit View Places Help
Sourcery CodeBench Lite for Xilinx GNU/Linux

Choose Install Set

- Welcome!
- Important Information
- Choose Install Set
- Choose Install Folder
- Add to PATH?
- Choose Link Folder
- Pre-Installation Summary
- Installing...
- Install Complete

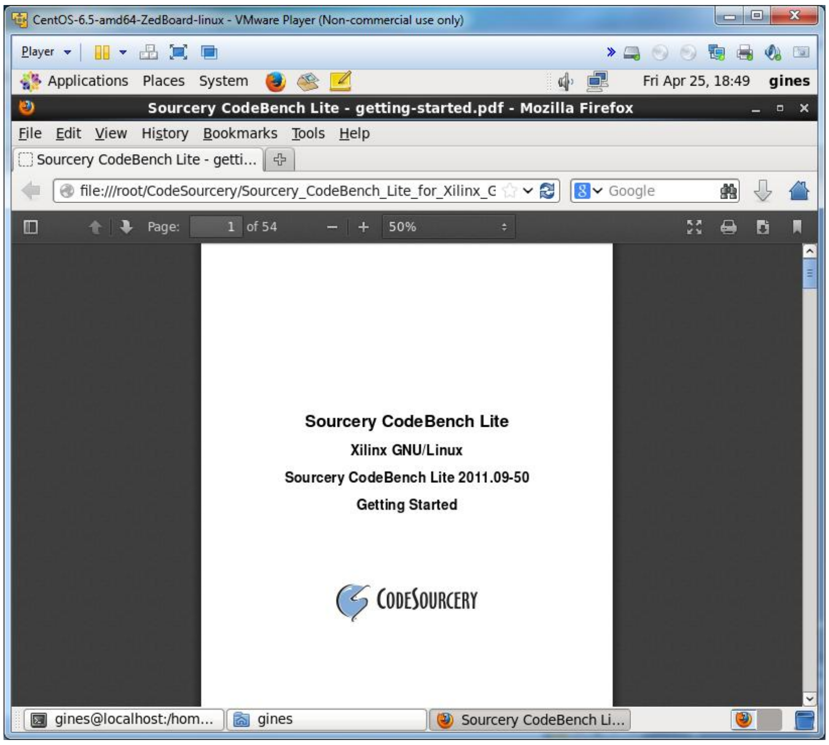
Typical
Install all Sourcery CodeBench components, including the Sourcery CodeBench IDE, command-line tools, and documentation.

Minimal
Install all Sourcery CodeBench components except the Sourcery CodeBench IDE.

Custom

Figure 199: Sourcery CodeBench - Choosing Install Set

Once installed, a Getting Started guide which contains useful information on use of the cross toolchain can be opened. Choose whether this document will be viewed and click “Next” and “Done”:


CentOS-6.5-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Player
Applications Places System
Fri Apr 25, 18:49 gines
Sourcery CodeBench Lite - getting-started.pdf - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Sourcery CodeBench Lite - getti...
file:///root/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_G
Google
Page: 1 of 54
50%
gines@localhost:/hom...
gines
Sourcery CodeBench Li...

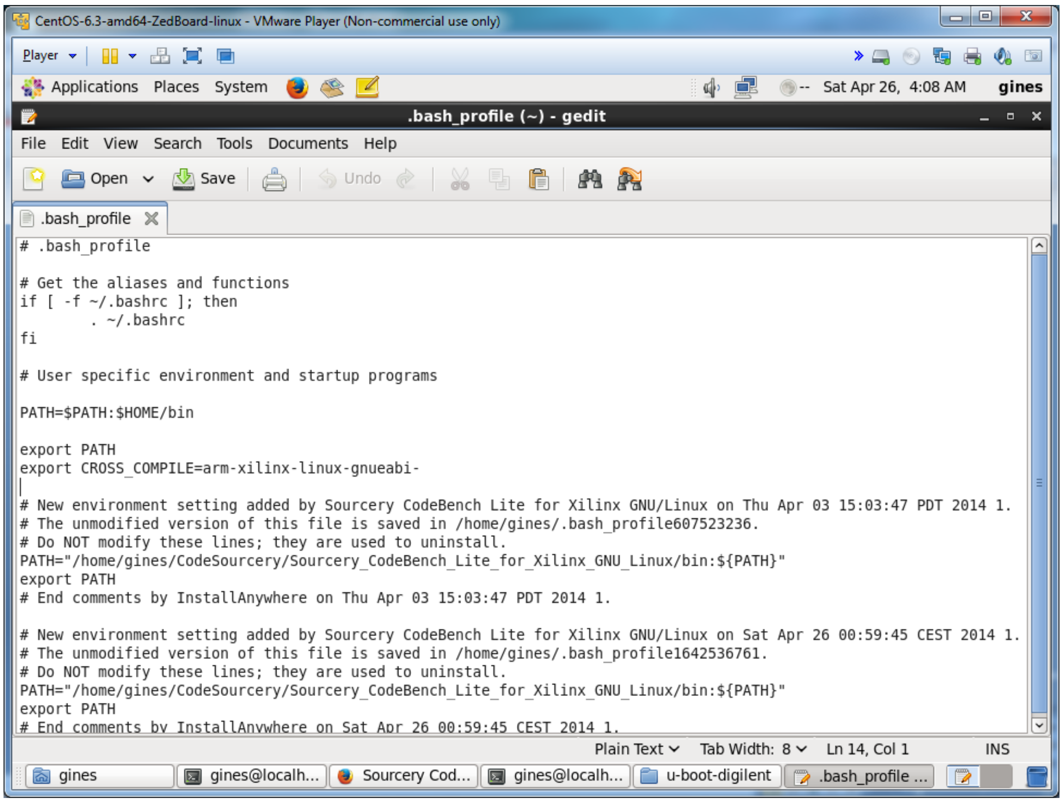
Sourcery CodeBench Lite
Xilinx GNU/Linux
Sourcery CodeBench Lite 2011.09-50
Getting Started

Figure 200: Getting Started Guide

Use the gedit text editor to open the bash shell user profile .bash_profile file found in the /home/gines/.bash_profile path. This file used to build the software package for a specific embedded target. The line “export CROSS_COMPILE=arm-xilinx-linux-gnueabi-” will be added to the bash shell user profile:

```
gedit .bash_profile
```

Command Window 32: Editing the “.bash_profile”


CentOS-6.3-amd64-ZedBoard-linux - VMware Player (Non-commercial use only)
Player
Applications Places System
Sat Apr 26, 4:08 AM gines
.bash_profile (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
.bash_profile
Plain Text Tab Width: 8 Ln 14, Col 1 INS
gines gines@localh... Sourcery Cod... gines@localh... u-boot-digilent .bash_profile ...

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-

# New environment setting added by Sourcery CodeBench Lite for Xilinx GNU/Linux on Thu Apr 03 15:03:47 PDT 2014 1.
# The unmodified version of this file is saved in /home/gines/.bash_profile607523236.
# Do NOT modify these lines; they are used to uninstall.
PATH="/home/gines/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_GNU_Linux/bin:${PATH}"
export PATH
# End comments by InstallAnywhere on Thu Apr 03 15:03:47 PDT 2014 1.

# New environment setting added by Sourcery CodeBench Lite for Xilinx GNU/Linux on Sat Apr 26 00:59:45 CEST 2014 1.
# The unmodified version of this file is saved in /home/gines/.bash_profile1642536761.
# Do NOT modify these lines; they are used to uninstall.
PATH="/home/gines/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_GNU_Linux/bin:${PATH}"
export PATH
# End comments by InstallAnywhere on Sat Apr 26 00:59:45 CEST 2014 1.
```

Figure 201: Final Aspect of the ".bash_profile" File

Save the changes and exit. The toolchain has already been added to the current user PATH environment variable. Pick up the updated user profile using the source command.

```
source .bash_profile
```

Command Window 33: Loading the ".bash_profile"

This completes the installation and configuration of the virtual machine. The virtual machine operating system can be suspended in any point. To do this, click on the window close X button in the upper right hand corner and click on the Suspend button when prompted. This will close the virtual machine window but it will also persist in the state that the CentOS desktop is left in so that the work can be resumed by re-launching the virtual machine.

A2.3.2. U-Boot Xilinx

Once again, open the "terminal" program. Xilinx has its modified sources for supporting their hardware. Therefore, the U-Boot sources used for this system are in following git repository which can be downloaded with the next command:

```
git clone git://github.com/Xilinx/u-boot-xlnx
```

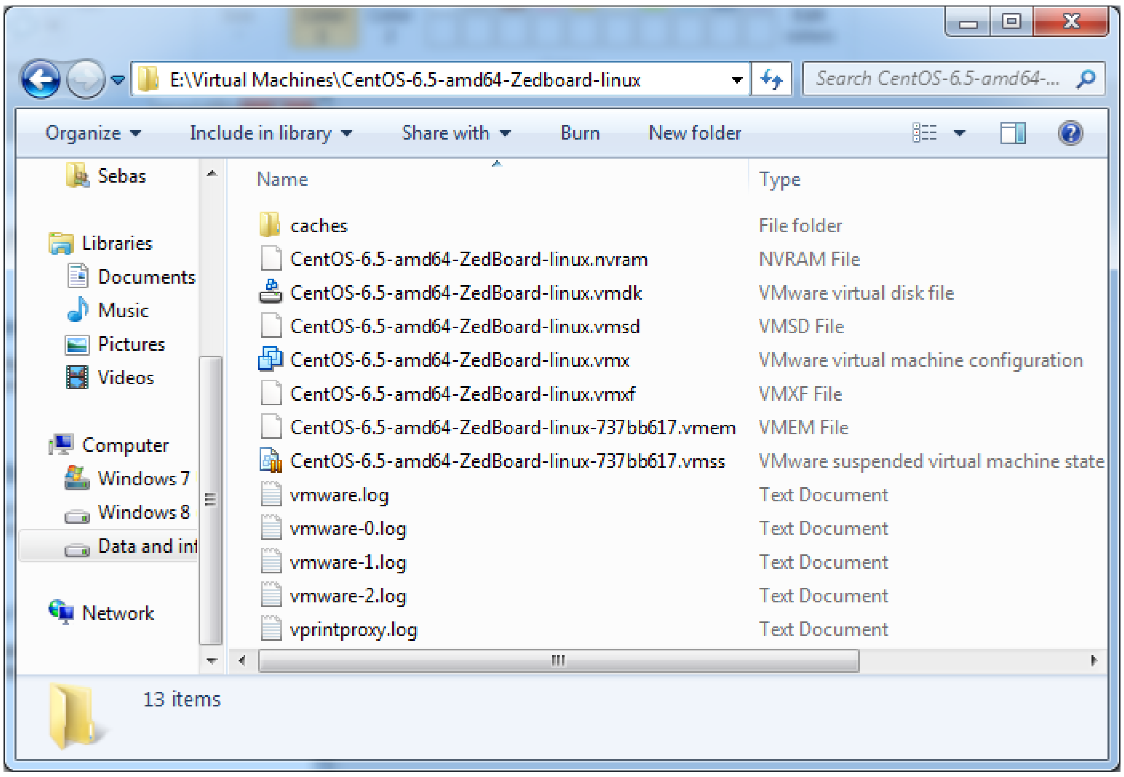
Command Window 34: Downloading the U-Boot Git Repository

A2.4. Copying the Virtual Machine

The virtual machine is completely configured. In the next sections, the specific programs for each Linux OS will be downloaded. Nevertheless, it is recommended to create another virtual machine. The first one will be used for the "Basic" Linux OS and the second one for the Ubuntu Linux.

In order to simplify the process, it will be faster to copy the first virtual machine at this point, because it is already configured. For that, search the installation directory of the last virtual machine. In this case, this directory is the following:

E:\Virtual Machines\CentOS-6.5-amd64-Zedboard-linux



E:\Virtual Machines\CentOS-6.5-amd64-Zedboard-linux

Search CentOS-6.5-amd64-...

Organize · Include in library · Share with · Burn · New folder

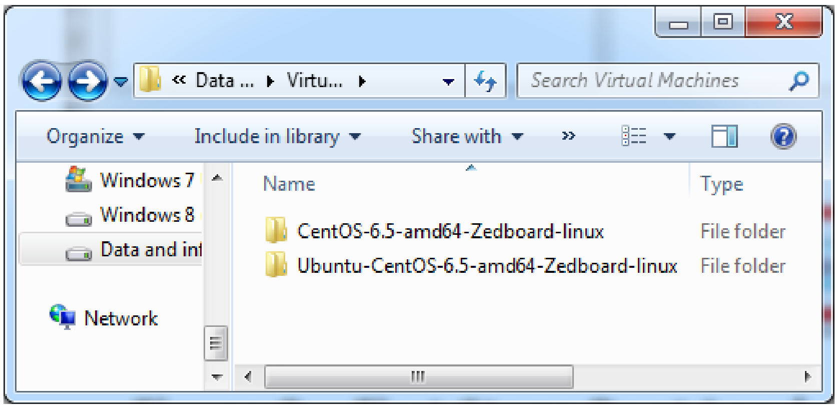
Sebas
Libraries
Documents
Music
Pictures
Videos
Computer
Windows 7
Windows 8
Data and inf
Network

Name	Type
caches	File folder
CentOS-6.5-amd64-ZedBoard-linux.nvram	NVRAM File
CentOS-6.5-amd64-ZedBoard-linux.vmdk	VMware virtual disk file
CentOS-6.5-amd64-ZedBoard-linux.vmsd	VMSD File
CentOS-6.5-amd64-ZedBoard-linux.vmx	VMware virtual machine configuration
CentOS-6.5-amd64-ZedBoard-linux.vmxf	VMXF File
CentOS-6.5-amd64-ZedBoard-linux-737bb617.vmem	VMEM File
CentOS-6.5-amd64-ZedBoard-linux-737bb617.vmss	VMware suspended virtual machine state
vmware.log	Text Document
vmware-0.log	Text Document
vmware-1.log	Text Document
vmware-2.log	Text Document
vprintproxy.log	Text Document

13 items

Figure 202: CentOS VM Installation Directory

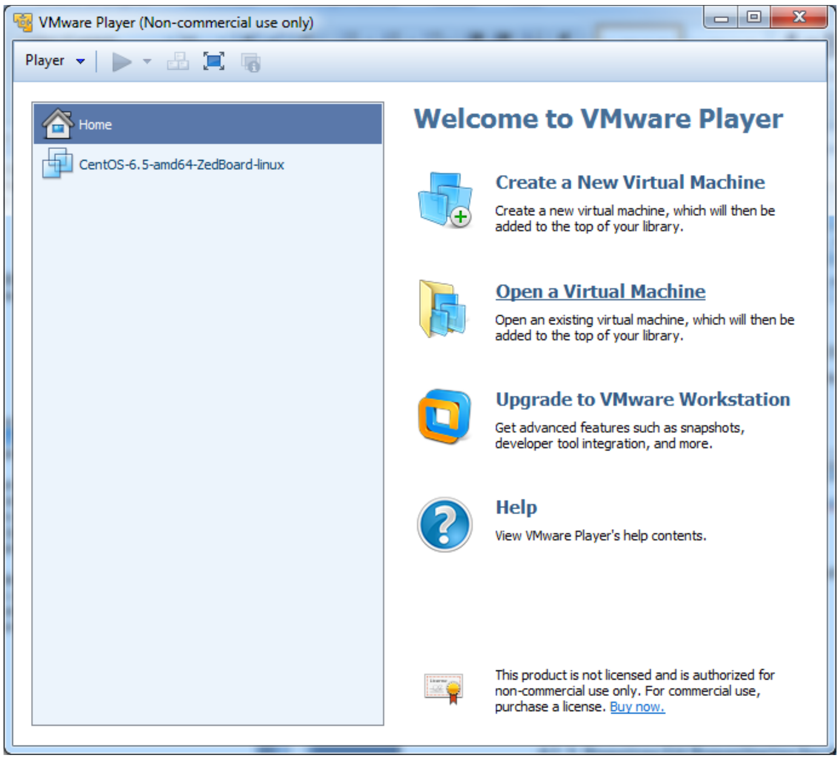
The whole folder must be copied and pasted. For instance, it can be pasted in the same folder, "Virtual Machines" with the name: "Ubuntu-CentOS-6.5-amd64-Zedboard-linux".



CentOS-6.5-amd64-Zedboard-linux File folder
Ubuntu-CentOS-6.5-amd64-Zedboard-linux File folder

Figure 203: CentOS Folder Copied and Renamed

Once the folder has been completely pasted, open again VMware and click in "Open a Virtual Machine".


VMware Player (Non-commercial use only)
Player
Home
CentOS-6.5-amd64-ZedBoard-linux

Welcome to VMware Player

Create a New Virtual Machine
Create a new virtual machine, which will then be added to the top of your library.

Open a Virtual Machine
Open an existing virtual machine, which will then be added to the top of your library.

Upgrade to VMware Workstation
Get advanced features such as snapshots, developer tool integration, and more.

Help
View VMware Player's help contents.

This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

Figure 204: Opening a VM in VMware Player

The .vmx file is available in the Ubuntu folder and it has to be chosen and opened.

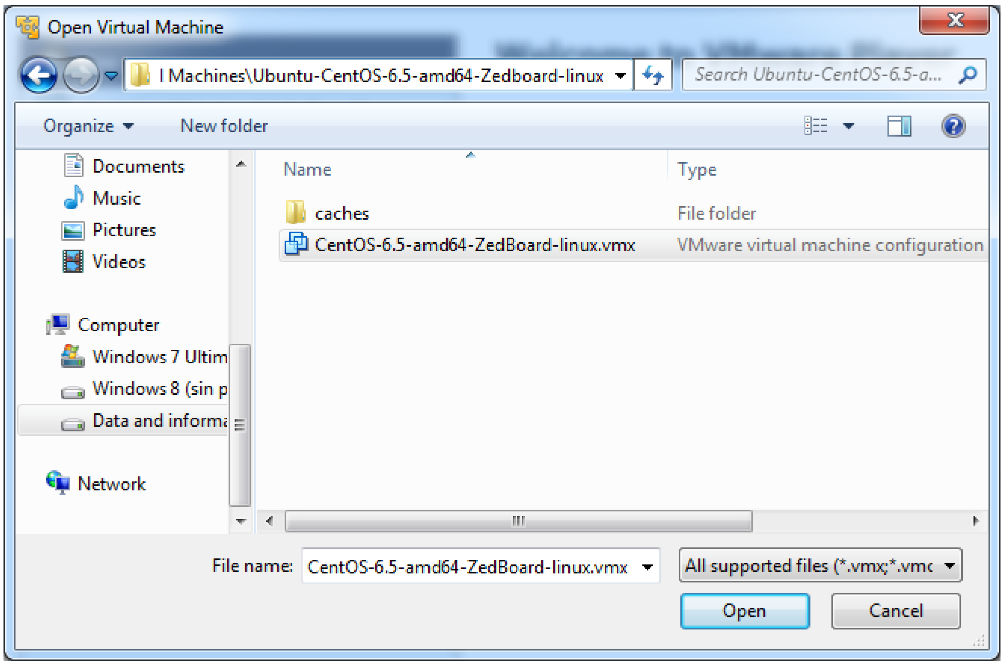

Open Virtual Machine
l Machines\Ubuntu-CentOS-6.5-amd64-Zedboard-linux
Search Ubuntu-CentOS-6.5-a...
Organize
New folder
Documents
Music
Pictures
Videos
Computer
Windows 7 Ultim
Windows 8 (sin p
Data and informa
Network
Name
Type
caches
File folder
CentOS-6.5-amd64-ZedBoard-linux.vmx
VMware virtual machine configuration
File name:
CentOS-6.5-amd64-ZedBoard-linux.vmx
All supported files (*.vmx;*.vmc
Open
Cancel

Figure 205: VM Opening Selection

Now, the two VM will appear with the same name in VMware Player. Select the file in the highest position, which will be the new added VM, and click in "Edit virtual machine settings".

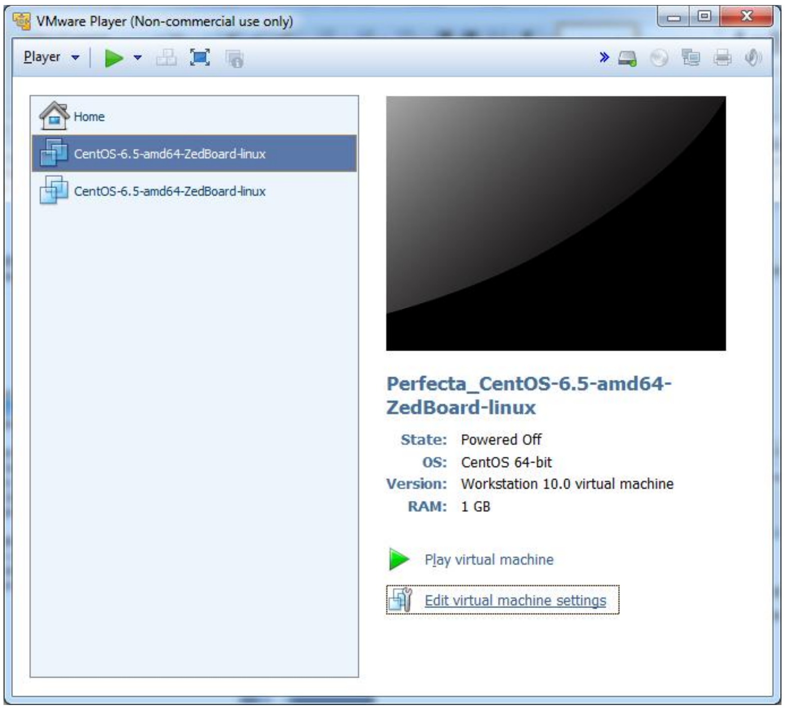
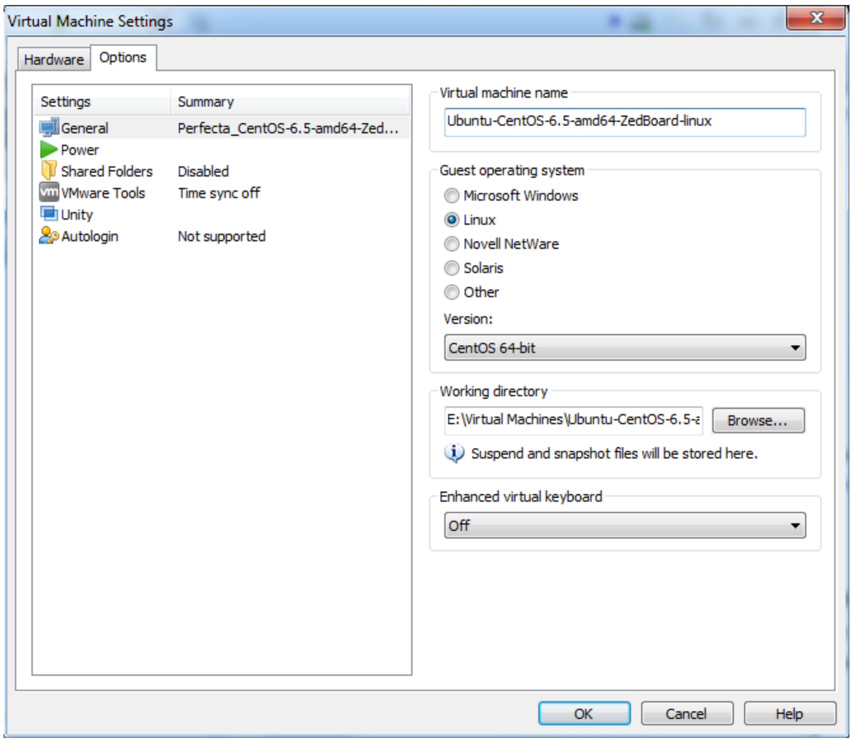

VMware Player (Non-commercial use only)
Player
Home
CentOS-6.5-amd64-ZedBoard-linux
CentOS-6.5-amd64-ZedBoard-linux
Perfecta_CentOS-6.5-amd64-ZedBoard-linux
State: Powered Off
OS: CentOS 64-bit
Version: Workstation 10.0 virtual machine
RAM: 1 GB
Play virtual machine
Edit virtual machine settings

Figure 206: VMware Player after Opening the VM

Click in the "Options" window, rename the "Virtual Machine Name" as "Ubuntu-CentOS-6.5-amd64-ZedBoard-linux" and "OK".



Settings	Summary
General	Perfecta_CentOS-6.5-amd64-Zed...
Power	
Shared Folders	Disabled
VMware Tools	Time sync off
Unity	
Autologin	Not supported

Virtual machine name: Ubuntu-CentOS-6.5-amd64-ZedBoard-linux

Guest operating system: Microsoft Windows [✓ Linux] Novell NetWare Solaris Other

Version: CentOS 64-bit

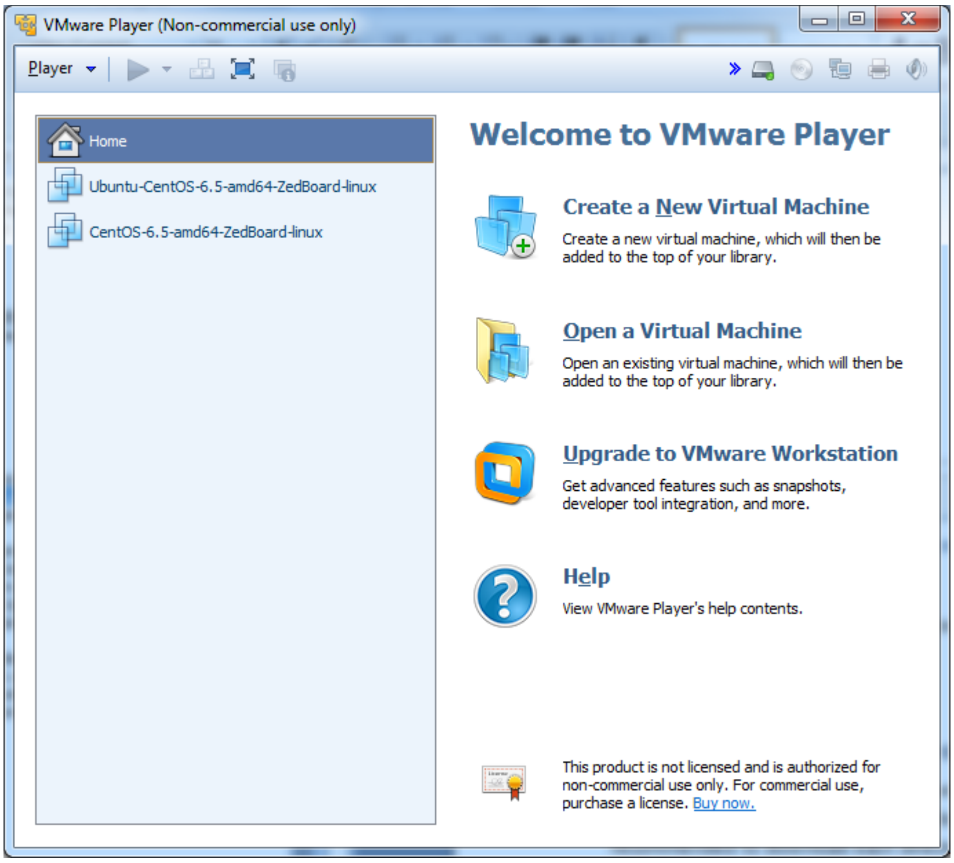
Working directory: E:\Virtual Machines\Ubuntu-CentOS-6.5-a

Suspend and snapshot files will be stored here.

Enhanced virtual keyboard: Off

Figure 207: VM Settings in VMware Player

Finally, both of the virtual machines are ready for their purpose.


VMware Player (Non-commercial use only)
Player
Home
Ubuntu-CentOS-6.5-amd64-ZedBoard-linux
CentOS-6.5-amd64-ZedBoard-linux

Welcome to VMware Player

Create a New Virtual Machine
Create a new virtual machine, which will then be added to the top of your library.

Open a Virtual Machine
Open an existing virtual machine, which will then be added to the top of your library.

Upgrade to VMware Workstation
Get advanced features such as snapshots, developer tool integration, and more.

Help
View VMware Player's help contents.

This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

Figure 208: VMware Ready for Be Used

A2.5. Required Git Repositories for the Custom Embedded Linux OS

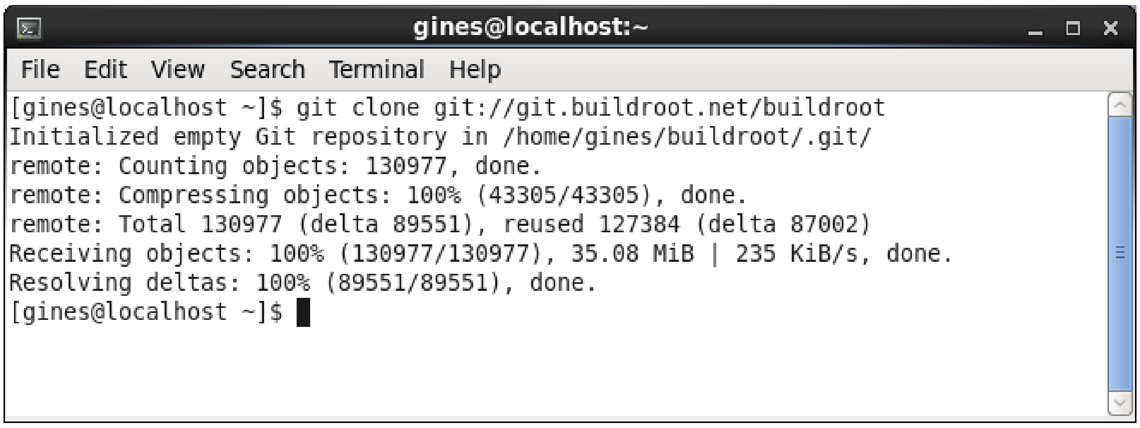
Note that too many “Terminal” windows can be opened at same time. Moreover, one of the following required files can be downloaded in each one at same time. Therefore, it is recommended to download each directory in one different window and go for a long coffee while all these directories are downloaded.

A2.5.1. BuildRoot

BuildRoot will be the main tool used to build the “Basic” Linux OS. It can be downloaded from its Git Repository:

```
git clone git://git.buildroot.net/buildroot
```

Command Window 35: Downloading BuildRoot



```
[gines@localhost ~]$ git clone git://git.buildroot.net/buildroot
Initialized empty Git repository in /home/gines/buildroot/.git/
remote: Counting objects: 130977, done.
remote: Compressing objects: 100% (43305/43305), done.
remote: Total 130977 (delta 89551), reused 127384 (delta 87002)
Receiving objects: 100% (130977/130977), 35.08 MiB | 235 KiB/s, done.
Resolving deltas: 100% (89551/89551), done.
[gines@localhost ~]$ 
```

Figure 209: BuildRoot Installation

A2.5.2. g++, qt4 and Development Tools

BuildRoot needs some programs to be executed without errors. The list of the required BuildRoot programs is available in the BuildRoot User Manual. First, login as root user and enter the password.

```
su
```

Command Window 36: Logging as Root User

Then, download and install the necessary programs with the following commands (write "Y" when it asks whether is ok in any case):

```
yum install gcc-c++
yum install qt qt-demos qt-designer qt4 qt4-designer
yum groupinstall 'Development Tools'
```

Command Window 37: Installing the Required Programs

A2.5.3. Linux-xlnx (optional)

Once again, open the "terminal" program. Xilinx has its modified sources for supporting their hardware, which can be downloaded from its own repository by typing the following command:

```
git clone git://github.com/Xilinx/linux-xlnx.git
```

Command Window 38: Downloading the Linux-xlnx Git Repository

A2.6. Required Git Repositories for the Ubuntu Linux OS

Remember that too many "Terminal" windows can be opened at same time. Therefore, it is strongly recommended to download each directory in one different "Terminal" window.

A2.6.1. Ubuntu Linux

A different Linux version from the previous Linux-xlnx repository is required for this Ubuntu Linux version. It can be downloaded by typing the following command, which will download the desired git repository but it will save it in the folder indicated in the own command, "ubuntu".

```
git://github.com/analogdevicesinc/linux.git ubuntu
```

Command Window 39: Downloading the Ubuntu Linux Git Repository

A2.6.2. GParted Partition Editor

GParted Partition Editor is a GNU program to create, modify and remove disk partitions. It will be used to create two partitions in the SD Card. It can be installed by typing the next command:

```
yum install gparted
```

Command Window 40: Downloading GParted Partition Editor

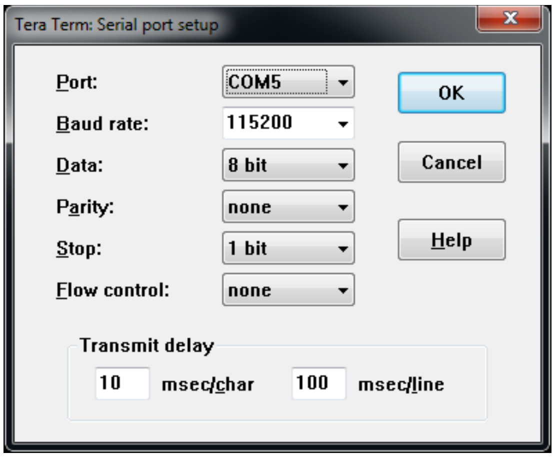
A2.2. Tera Term

Tera Term is an open-source terminal emulator program. It emulates different types of computer terminals, from DEC VT100 to DEC VT382. It supports telnet, SSH 1 & 2 and serial port connections. It also has a built-in macro scripting language (supporting Oniguruma regular expressions) and a few other useful plugins.

Therefore, Tera Term will be used to interchange information between the ZedBoard and any computer. It can be downloaded for free. For instance, in its official website:

http://en.sourceforge.jp/projects/ttssh2/

Once Tera Term is installed, to configure baud rate settings, open the Serial Port Setup window from "Setup" and "Serial port..." in the menu selection. Configure as the following figure. Notice that the Port will depend of each computer and must be checked.



Port: COM5
Baud rate: 115200
Data: 8 bit
Parity: none
Stop: 1 bit
Flow control: none
Transmit delay
10 msec/char
100 msec/line

Figure 210: Tera Term – Serial Port Setup