Universidad Politécnica de Madrid

Escuela Universitaria de Ingeniería Técnica de Telecomunicación

PROYECTO FIN DE CARRERA

Development of Embedded Linux Applications Using ZedBoard

Autor: Álvaro Bustos Benayas
Tutor: Mariano Ruíz González

Julio 2013
The purpose of this project is to create an integration guide of embedding a Linux Operating System on ZedBoard development platform, based on Xilinx's Zynq-7000 All Programmable System on Chip which contains a dual core ARM Cortex-A9 and a 7 Series FPGA Artix-7.

The integration process has been structured in four chapters according to the logic generation of the different parts that compose the embedded system.

With the intention of automating the generation process of the complete Linux distribution specific for ZedBoard platform, BuildRoot development platform is used.

Once the embedding process finished, it was decided to add to the system the required functionalities for adding support for IEEE 1588 Standard for Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, through a user space Linux program which implements the protocol. That PTP user space implementation program has been cross-compiled, executed on target and tested for evaluating the functionalities added.
“Always believe in your dreams, because if you don’t, you’ll still have hope.”

Mahatma Gandhi, 1924.
I would like to specially thank to Mariano for bringing me the opportunity of working in this project, sharing his knowledge with me and helping me becoming a more competent person. Also I would like to thank to all my colleagues of the 8111 for sharing their knowledge and to liven up the gone by hours in the lab.

Thanks to the people that loves me and cares about me for their support and for accepting my apologizes for being absentee for so long, to my patient parents for encouraging me to accomplish my goals; to the people that belays me and makes life a little more spicy; and also to the university-family set some years ago for making me enjoying the brief relaxation moments of the day.
Abstract

The purpose of this document is to create a modest integration guide for embedding a Linux Operating System on ZedBoard development platform, based on Xilinx’s Zynq-7000 All Programmable System on Chip which contains a dual core ARM Cortex-A9 and a 7 Series FPGA Artix-7.

The integration process has been structured in four chapters according to the logic generation of the different parts that compose the embedded system.

With the intention of automating the generation process of a complete Linux distribution specific for ZedBoard platform, BuildRoot development platform it is used.

Once the embedding process finished, it was decided to add to the system the required functionalities for adding support for IEEE1588 Standard for Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, through a user space Linux program which implements the protocol. That PTP user space implementation program has been cross-compiled, executed on target and tested for evaluating the functionalities added.
El propósito de este documento es crear una modesta guía de integración de un sistema operativo Linux para la plataforma de desarrollo ZedBoard, basada en un System on Chip del fabricante Xilinx llamado Zynq-7000. Este System on Chip está compuesto por un procesador de doble núcleo ARM Cortex-A9 y una FPGA de la Serie 7 equiparable a una Artix-7.

El proceso de integración se ha estructurado en cuatro grandes capítulos que se rigen según el orden lógico de generación de las distintas partes por las que el sistema empotrado está compuesto.

Con el ánimo de automatizar el proceso de creación de una distribución de Linux específica para la plataforma ZedBoard, se ha utilizado la plataforma de desarrollo BuildRoot.

Una vez terminado el proceso de integración del sistema empotrado, se procedió a dar dotar al sistema de las funcionalidades necesarias para dar soporte al estándar de sincronización de relojes en redes de área local, PTP IEEE1588, a través de una implementación del mismo en un programa de lado de usuario el cual ha sido compilado, ejecutado y testado para evaluar el correcto funcionamiento de las funcionalidades añadidas.
# Table of Contents

Chapter 1: Introduction to Embedded systems, the ZedBoard Platform ................................................................. 1

1.1 Embedded systems ........................................................................................................................................... 1

1.2 Avnet ZedBoard ........................................................................................................................................... 2
  1.2.1 Block Diagram ......................................................................................................................................... 3

1.3 Xilinx Zynq-7000 SoC ............................................................................................................................... 4
  1.3.1 Zynq-7000 Processor Overview ........................................................................................................... 5
  1.3.2 Programmable Logic Overview ........................................................................................................... 6

1.4 Summary of the Next Chapters ................................................................................................................... 6

Chapter 2: Development Tools ............................................................................................................................ 9

2.1 Design Flow for Zynq-7000 ....................................................................................................................... 9

2.2 Xilinx Design Tools ................................................................................................................................... 12
  2.2.1 Plan Ahead ......................................................................................................................................... 12
  2.2.2 XPS .................................................................................................................................................. 19
  2.2.3 Design Constraints .......................................................................................................................... 28
  2.2.4 Synthesis, Implementation and Bitstream Generation. ..................................................................... 30
  2.2.5 SDK .................................................................................................................................................. 33
  2.2.6 Installing Device Tree Generator BSP ............................................................................................. 36

2.3 GNU Tools .................................................................................................................................................. 37
  2.3.1 Sourcery Codebench Lite .................................................................................................................. 37
  2.3.2 BuildRoot ......................................................................................................................................... 41
  2.3.3 BusyBox ......................................................................................................................................... 41
  2.3.4 Digilent Sources ............................................................................................................................ 42

Chapter 3: Linux on ZedBoard ............................................................................................................................ 43

3.1 Overview .................................................................................................................................................... 43

3.2 Stage 0: BootROM ................................................................................................................................. 43

3.3 Stage 1: First Stage Boot-Loader ........................................................................................................... 45
  3.3.1 Building FSBL .................................................................................................................................... 45

3.4 Stage 2: Second Stage Boot-Loader, U-Boot ....................................................................................... 51
  3.4.1 Building the U-boot Bootloader ..................................................................................................... 52
5.4.3 Communication paradigm between the two driver layers ................................................. 139

5.5 Xilinx Ethernet Driver, The xilinx_emacps ................................................................................. 141

5.5.1 Linux Support for IEEE1588 in the Xilinx Ethernet Driver ................................................. 141

5.5.2 PHC in the Xilinx Ethernet Driver ...................................................................................... 143

5.5.3 Testing the Driver .............................................................................................................. 158

5.6 User Space Program .................................................................................................................. 160

5.6.1 Getting Sources ................................................................................................................. 160

5.6.2 Compiling Sources ............................................................................................................. 160

5.6.3 Debugging the User Space Program, GDB Server ............................................................. 163

Chapter 6: Conclusions and Future Work ................................................................................................. 171

6.1 Summary ................................................................................................................................... 171

6.1.1 Customization, Construction and Integration of a Linux OS for ZedBoard Platform........ 171

6.1.2 Zynq-7000 Support for PTP IEEE1588 Standard ................................................................. 172

6.2 Future Work .............................................................................................................................. 172

Chapter 7: Bibliography ............................................................................................................................ 175
Table of Figures

Figure 1 ZedBoard Block Diagram ................................................................................................................. 3
Figure 2 Zynq-7000 AP SoC Overview ........................................................................................................... 5
Figure 3 ZedBoard Linux Boot Medium ......................................................................................................... 7
Figure 4 Zynq-7000 SoC Embedded Design Flow Overview ...................................................................... 11
Figure 5 PlanAhead Creating A Project 1 ..................................................................................................... 13
Figure 6 PlanAhead Creating A Project 2 ..................................................................................................... 13
Figure 7 PlanAhead Creating A Project 3 ..................................................................................................... 14
Figure 8 PlanAhead Creating A Project 4, Target Device xc7z020clg484-1 ................................................ 15
Figure 9 PlanAhead Creating A Project 5, Project Summary ...................................................................... 15
Figure 10 PlanAhead Creating A Project 6, PA Main Window .................................................................... 16
Figure 11 PlanAhead Adding Embedded Sources 1 .................................................................................... 16
Figure 12 PlanAhead Adding Embedded Sources 2 .................................................................................... 17
Figure 13 PlanAhead Adding Embedded Sources 3 .................................................................................... 17
Figure 14 PlanAhead Adding Embedded Sources 4 .................................................................................... 18
Figure 15 PlanAhead Adding Embedded Sources 5 .................................................................................... 19
Figure 16 Non-configured XPS .................................................................................................................... 20
Figure 17 Exploring IP Catalogue ................................................................................................................ 20
Figure 18 Adding Processing System 7 ..................................................................................................... 21
Figure 19 XPS PS7 Bus Interfaces Tab ......................................................................................................... 21
Figure 20 XPS Zynq-7000 PS7 System Assembly View ............................................................................. 22
Figure 21 XPS Import ZedBoard Board Definition File ............................................................................. 23
Figure 22 XPS MIO Configurations 1 ........................................................................................................... 23
Figure 23 XPS MIO Configurations 2 ........................................................................................................... 24
Figure 24 PS PLL Switching Characteristics, extracted from [4] Zynq-7000 DC and AC Switching Characteristics Data Sheet table 19. ........................................................................................................... 25
Figure 25 XPS Clock Generation Block, Clock Wizard ............................................................................. 26
Figure 26 XPS Processing System 7 DDR Configuration ........................................................................... 27
Figure 27 DDR3 Worksheet Calculations for ZedBoard ............................................................................. 28
Figure 28 XPS Processing System 7 DDR Configuration Complete ............................................................. 28
Figure 29 PlanAhead Add Sources, Creating Constrains ........................................................................ 29
Figure 30 PlanAhead Adding UCF File ....................................................................................................... 29
Figure 31 Adding a Top Level Module for the Embedded System Design .................................................. 30
Figure 32 ISE FPGA Design Flow Overview ............................................................................................. 31
Figure 33 PlanAhead Generate Bitstream .................................................................................................. 33
Figure 34 ZedBoard Linux Boot Medium ................................................................................................... 35
Figure 35 PlanAhead, Export Hardware for SDK ..................................................................................... 35
Figure 36 PlanAhead, Exporting Options .................................................................................................. 36
Figure 37 Adding Device Tree BSP Support in Xilinx EDK ....................................................................... 37
Figure 38 Sourcery Codebench Lite for Xilinx GNU/Linux Launching the Installer ..................................... 38
Figure 81 Buildroot Filesystem Image type and options .......................................................... 88
Figure 82 Buildroot Kernel Configuration ............................................................................. 89
Figure 83 Buildroot Kernel binary format ............................................................................. 89
Figure 84 Interactive Curses-based Kernel Configurator ....................................................... 91
Figure 85 Linux kernel menuconfig 1, General Setup ........................................................ 92
Figure 86 Linux kernel menuconfig 2, General Setup ........................................................ 92
Figure 87 Linux kernel menuconfig 3, General Setup ........................................................ 92
Figure 88 Linux kernel menuconfig 4, Enable Loadable Module Support .......................... 93
Figure 89 Linux kernel menuconfig 5, Block Layer IO Schedulers .................................... 93
Figure 90 Linux kernel menuconfig 6, System Type ............................................................. 93
Figure 91 Linux kernel menuconfig 7, System Type ............................................................. 93
Figure 92 Linux kernel menuconfig 8, System Type Xilinx Specific Options ..................... 94
Figure 93 Linux kernel menuconfig 9, Kernel Features ...................................................... 94
Figure 94 Linux kernel menuconfig 10, Kernel Features ................................................... 94
Figure 95 Linux kernel menuconfig 11, Boot Options ....................................................... 95
Figure 96 Linux kernel menuconfig 12, FPU .................................................................... 95
Figure 97 Linux kernel menuconfig 13, Userspace Binary Formats .................................. 95
Figure 98 Linux kernel menuconfig 14, Power Management ............................................ 95
Figure 99 Linux kernel menuconfig 15, Networking Options ........................................... 96
Figure 100 Linux kernel menuconfig 16, Networking Options .......................................... 96
Figure 101 Linux kernel menuconfig 17, Networking Options ........................................... 97
Figure 102 Linux kernel menuconfig 18, Device Drivers .................................................. 97
Figure 103 Linux kernel menuconfig 19, Device Drivers .................................................. 98
Figure 104 Linux kernel menuconfig 20, Device Drivers .................................................. 98
Figure 105 Linux kernel menuconfig 21, Generic Driver Options ................................... 98
Figure 106 Linux kernel menuconfig 22, Connector Between Userspace and Kernelspace .. 99
Figure 107 Linux kernel menuconfig 23, MTD Support ..................................................... 99
Figure 108 Linux kernel menuconfig 24, MTD Support ..................................................... 99
Figure 109 Linux kernel menuconfig 25, Device Tree Support ........................................ 99
Figure 110 Linux kernel menuconfig 26, Block Devices .................................................... 100
Figure 111 Linux kernel menuconfig 27, SCSI Device Support ......................................... 100
Figure 112 Linux kernel menuconfig 28, Network Device Support ................................ 100
Figure 113 Linux kernel menuconfig 29, PHY Device and Infrastructure ....................... 101
Figure 114 Linux kernel menuconfig 30, Ethernet Driver Support ................................ 101
Figure 115 Linux kernel menuconfig 31, Ethernet Driver Support ................................ 101
Figure 116 Linux kernel menuconfig 32, Input Device Support ....................................... 102
Figure 117 Linux kernel menuconfig 33, Character Drivers ............................................. 102
Figure 118 Linux kernel menuconfig 34, I2C Support ....................................................... 102
Figure 119 Linux kernel menuconfig 35, SPI Support ..................................................... 103
Figure 120 Linux kernel menuconfig 36, GPIO Support .................................................. 103
Figure 121 Linux kernel menuconfig 37, WDT Support .................................................. 103
Figure 122 Linux kernel menuconfig 38, USB Support .................................................... 104
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>165</td>
<td>Busybox menuconfig 30, Networking Utilities</td>
<td>120</td>
</tr>
<tr>
<td>166</td>
<td>Busybox menuconfig 31, Networking Utilities</td>
<td>121</td>
</tr>
<tr>
<td>167</td>
<td>Busybox menuconfig 32, Networking Utilities</td>
<td>121</td>
</tr>
<tr>
<td>168</td>
<td>Busybox menuconfig 33, Networking Utilities</td>
<td>121</td>
</tr>
<tr>
<td>169</td>
<td>Busybox menuconfig 34, Networking Utilities</td>
<td>122</td>
</tr>
<tr>
<td>170</td>
<td>Busybox menuconfig 35, Process Utilities</td>
<td>122</td>
</tr>
<tr>
<td>171</td>
<td>Busybox menuconfig 36, Process Utilities</td>
<td>122</td>
</tr>
<tr>
<td>172</td>
<td>Busybox menuconfig 37, Shells</td>
<td>123</td>
</tr>
<tr>
<td>173</td>
<td>Busybox menuconfig 38 System Logging utilities</td>
<td>123</td>
</tr>
<tr>
<td>174</td>
<td>Interactive Curses-based uClibc Configurator</td>
<td>124</td>
</tr>
<tr>
<td>175</td>
<td>uClibc menuconfig 1, Features and Options</td>
<td>124</td>
</tr>
<tr>
<td>176</td>
<td>uClibc menuconfig 2, Linux Kernel Header Location</td>
<td>124</td>
</tr>
<tr>
<td>177</td>
<td>uClibc menuconfig 3, General Library Settings</td>
<td>125</td>
</tr>
<tr>
<td>178</td>
<td>uClibc menuconfig 4, General Library Settings</td>
<td>125</td>
</tr>
<tr>
<td>179</td>
<td>uClibc menuconfig 5, General Library Settings</td>
<td>125</td>
</tr>
<tr>
<td>180</td>
<td>uClibc menuconfig 6, Advanced Library Settings</td>
<td>126</td>
</tr>
<tr>
<td>181</td>
<td>uClibc menuconfig 7, Network Support</td>
<td>126</td>
</tr>
<tr>
<td>182</td>
<td>uClibc menuconfig 8, String and Stdio Support</td>
<td>126</td>
</tr>
<tr>
<td>183</td>
<td>uClibc menuconfig 9, String and Stdios Support</td>
<td>127</td>
</tr>
<tr>
<td>184</td>
<td>uClibc menuconfig 10, Big and Tall</td>
<td>127</td>
</tr>
<tr>
<td>185</td>
<td>uClibc menuconfig 11, Library Installation Options</td>
<td>127</td>
</tr>
<tr>
<td>186</td>
<td>uClibc menuconfig 12, Security Options</td>
<td>127</td>
</tr>
<tr>
<td>187</td>
<td>uClibc menuconfig 13, Development and Debugging Options</td>
<td>128</td>
</tr>
<tr>
<td>188</td>
<td>uClibc menuconfig 14, Cross-compiling Toolchain Prefix</td>
<td>128</td>
</tr>
<tr>
<td>189</td>
<td>Buildroot Default Skeleton Rootfilesystem</td>
<td>128</td>
</tr>
<tr>
<td>190</td>
<td>Zyq-7000 Gigabit Ethernet Controller System View Point, image extracted from [1]</td>
<td>130</td>
</tr>
<tr>
<td>191</td>
<td>Kernel PTP 1588 Clock Support</td>
<td>132</td>
</tr>
<tr>
<td>192</td>
<td>Enable Linux Kernel PTP 1588 Clock Support</td>
<td>133</td>
</tr>
<tr>
<td>193</td>
<td>Help of the PTP Clock Support Configuration</td>
<td>134</td>
</tr>
<tr>
<td>194</td>
<td>PTP Linux Driver Layers</td>
<td>136</td>
</tr>
<tr>
<td>195</td>
<td>Device Driver Data Structures Diagram</td>
<td>141</td>
</tr>
<tr>
<td>196</td>
<td>Generate Hardware Timestamps Support for xilinx_emacps Ethernet Driver</td>
<td>142</td>
</tr>
<tr>
<td>197</td>
<td>Linux Description of Hardware Timestamps Support</td>
<td>143</td>
</tr>
<tr>
<td>198</td>
<td>1588 Timer Increment Register Details</td>
<td>150</td>
</tr>
<tr>
<td>199</td>
<td>Induced Diagram Model of 1588 Timer Register Adjust</td>
<td>151</td>
</tr>
<tr>
<td>200</td>
<td>WolframAlpha Diophantine Equation Resolve for the Maximum Lower Value</td>
<td>153</td>
</tr>
<tr>
<td>201</td>
<td>WolframAlpha Diophantine Equation Resolve for the Maximum Lower Value Truncated</td>
<td>154</td>
</tr>
<tr>
<td>202</td>
<td>WolframAlpha Diophantine Equation Resolve for the Maximum Upper Value</td>
<td>155</td>
</tr>
<tr>
<td>203</td>
<td>WolframAlpha Diophantine Equation Resolve for the Maximum Upper Value Truncated</td>
<td>156</td>
</tr>
<tr>
<td>204</td>
<td>Eclipse Program Development Cross Building Platform, Cross Settings</td>
<td>164</td>
</tr>
<tr>
<td>205</td>
<td>Eclipse Program Development Cross Building Platform, Including Linux User Libraries</td>
<td>165</td>
</tr>
<tr>
<td>206</td>
<td>Eclipse Program Development Cross Building Platform, Including BuildRoot Host Tools</td>
<td>166</td>
</tr>
</tbody>
</table>
Figure 207 BuildRoot Adding Support for GDBServer ................................................................. 167
Figure 208 Eclipse Program Development Cross Building Platform, Creating a New Launch Configuration for Debugging.......................................................... 168
Figure 209 Eclipse Program Development Cross Building Platform, Launching Modes....................... 168
Figure 210 Eclipse Program Development Cross Building Platform, Set the Path of the gdbserver .... 169
Figure 211 Eclipse Program Development Cross Building Platform, Configure the TCP Connection .... 170
<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>zynq_zed.h</td>
<td>55</td>
</tr>
<tr>
<td>2</td>
<td>ZedBoard DTS Extract</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td>Bootargs in chosen node</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Top-level Source README File for every Linux Kernel</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>Makefile PTP Clock Support</td>
<td>134</td>
</tr>
<tr>
<td>6</td>
<td>ptp_clock_kernel.h (Copyright © 2010 OMICRON electronics GmbH)</td>
<td>136</td>
</tr>
<tr>
<td>7</td>
<td>Definition of PTP Hardware Clock Structure, Extract of ptp_private.h (Copyright © 2010 OMICRON electronics GmbH)</td>
<td>138</td>
</tr>
<tr>
<td>8</td>
<td>Extract of ptp_clock.c Class Driver</td>
<td>139</td>
</tr>
<tr>
<td>9</td>
<td>Extract of PTP Hardware Clock Driver Implementation</td>
<td>140</td>
</tr>
<tr>
<td>10</td>
<td>Makefile for Xilinx Network Drivers</td>
<td>142</td>
</tr>
<tr>
<td>11</td>
<td>ptp_clock_info Structure Declared in ptp_clock_kernel.h</td>
<td>144</td>
</tr>
<tr>
<td>12</td>
<td>Clock Structure for Management, Host in xilinx_emacps</td>
<td>144</td>
</tr>
<tr>
<td>13</td>
<td>Set up of the PHC Structure</td>
<td>144</td>
</tr>
<tr>
<td>14</td>
<td>Registration of the PHC</td>
<td>145</td>
</tr>
<tr>
<td>15</td>
<td>Obtainning the Virtual Address of the Ethernet Controller</td>
<td>145</td>
</tr>
<tr>
<td>16</td>
<td>xemacps_read_ptp_clock and xemacps_get_hwticks Functions for Supporting PTP Hardware Control</td>
<td>145</td>
</tr>
<tr>
<td>17</td>
<td>xemacps_write_ptp_clock and xemacps_set_ptp_hwticks Functions for Supporting PTP Hardware Control</td>
<td>146</td>
</tr>
<tr>
<td>18</td>
<td>Driver Method gettime</td>
<td>147</td>
</tr>
<tr>
<td>19</td>
<td>Driver Method settime</td>
<td>148</td>
</tr>
<tr>
<td>20</td>
<td>Driver Method adjtime</td>
<td>148</td>
</tr>
<tr>
<td>21</td>
<td>Driver Method enable</td>
<td>149</td>
</tr>
<tr>
<td>22</td>
<td>Driver Method adjfreq</td>
<td>156</td>
</tr>
<tr>
<td>23</td>
<td>testptp Makefile</td>
<td>159</td>
</tr>
<tr>
<td>24</td>
<td>linuxptp Makefile</td>
<td>160</td>
</tr>
</tbody>
</table>
## Table of Tables

Table 1 Devicetree Bootargs ................................................................. 69
Table 2 Zynq-7000 Register Support for IEEE1588 ........................................ 131
Chapter 1: Introduction to Embedded systems, the ZedBoard Platform

1.1 Embedded systems

Embedded systems are computer-based systems designed for specific functions rather than be general-purpose. Since one of the early first embedded systems developed by IBM for the Gemini Project\(^1\) founded on an on-board computer integrated with other spacecraft systems, up to now, that we have computers in our ovens, cell phones, digital music players, videogame consoles, etc. we can imagine how this devices evolved.

Embedded systems are not always standalone devices; many of them consist of small parts within a larger device that serves a more general purpose. Such as an embedded fuel injection control in an automobile provides a specific subsystem function.

Embedded systems rove from not user interface at all to complex graphical user interfaces depending on the purpose to which are made for. Typically, they have at least a basic interface for the developer or for its maintenance for example a Serial Communication Interface port to talk with the outside world or a LCD.

One of the most important issues that embedded designers have to take care is the overburden of the system. The system bottlenecks imply the increment of the data latency, delay interrupt handling and lower data throughput among others. Parallel Processing is efficient for critical system performance but a central controller and memory management is needed; one possible solution for parallel processing can be achieved within a FPGA.

One traditional solution is building a discrete hybrid system, a microcontroller put together with a FPGA, that unites the best of both worlds but with some exceptions like: the bandwidth between the two IC’s is limited; increased power consumption; increased PCB size and layout complexity; performing limit of the interface that connects the microprocessor and the FPGA among others.

The demands of today’s technology results in the fusion of a processing system and a programmable logic in a single device, that’s where the Zynq-7000 All Programmable System on Chip appears. Zynq-7000 is not an FPGA with an embedded PowerPC\(^2\), 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 with a wide

\(^1\) NASA Gemini Project [25]

\(^2\) PowerPC architecture in Xilinx devices [33]
range of hard-core interface functions like high performance I/Os, Gigabit Transceivers, high throughput AXI (Advanced eXtensively Interface) up to three thousand PS to PL connections

This two-chip combo All Programmable SoC will reduce the cost, size, complexity and power consumption of the system; at the same time increasing the system performance.

1.2 Avnet ZedBoard

ZedBoard is intended to be a community development platform evaluation and development board based on the Xilinx Zynq-7000 All Programmable System on Chip. Combining a 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. Key Features provided by ZedBoard consist of:

- Xilinx XC7Z020-1CGL484CES Zynq-7000 AP SoC
  - Up to 667 MHz operation
  - NEON Processing/FPU Engines
- Memory512 MB DDR3 memory
  - 256 Mb Quad SPI Flash
  - 512 MB DDR3 (128 x 32)
  - 4GB SD Card
- Connectivity
  - 10/100/1000 Ethernet
  - USB-JTAG Programming
  - SD Card
  - USB 2.0 FS USB-UART bridge
  - Five Digilent Pmod headers
  - FMC (Low Pin Count) connector
  - USB OTG 2.0(Device/Host/OTG)
  - Two Reset Buttons (1 PS, 1 PL)
  - Seven Push Buttons (2 PS, 5 PL)
  - Eight switches (PL)
  - Nine User LEDs (1 PS, 8 PL)
  - ARM Debug Access Port (DAP)
  - Xilinx XADC Header
- On-board Oscillators
  - 33.333 MHz (PS)
  - 100 MHz (PL)
- Display/Audio
  - HDMI Output
  - VGA (12-bit Colour)
Introduction to Embedded systems, the ZedBoard Platform

- 128x32 OLED Display
- Audio Line-in, Line-out, headphone, microphone
- Power
  - 12V DC input @ 3.0 A (Max)
- Dimensions
  - Length 16 cm, Width 13.5 cm

1.2.1 Block Diagram

Figure 1 shows the block diagram of the different peripheral presented in the ZedBoard and how the connections to the Zynq-7000 SoC are made.

![Figure 1 ZedBoard Block Diagram](image-url)
1.3 Xilinx Zynq-7000 SoC

The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture. These products integrate a feature-rich dual-core ARM Cortex-A9 MPCore based processing system (PS) and Xilinx programmable logic (PL) in a single device. 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.

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. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices.

Figure 2 illustrates the functional blocks of the Zynq-7000 AP SoC. The PS and the PL are on separate power domains, enabling the user of these devices to power down the PL for management if required.

The processors in the PS always boot first. The PL can be configured as part of the boot process or configured at some point in the future. Additionally, the PL can be completely reconfigured or used with partial, dynamic reconfiguration (PR). PR allows configuration of a portion of the PL.
1.3.1 Zynq-7000 Processor Overview

1.3.1.1 ARM Cortex-A9 Processor and L1 Caches
The main features of the ARM core are the following

- ARM Cortex-A9 processor key features
  - 2.5DMIPS/MHz per core
  - Up to 1GHz operation

- L1 Cache key features
  - 32KB instruction and 32KB Data cache
  - Cache line length is 32 bytes
  - All L1 caches support parity

1.3.1.2 L2 Cache Controller
ARM-based processing needs specific high performance cache controller. The features are:

- High performance L2 cache controller
  - 512KB of cache
  - Supports lockdown by line for routines that might need low latency
  - Parity support

- L2 cache controller AXI interfaces
  - One AXI master interface for the DDR controller
  - One AXI master interface for all slave devices in the PL and PS
  - One AXI slave interface for Snoop Control Unit

1.3.1.3 Snoop Control Unit (SCU)
SCU connects two Cortex-A9 processors to the system memory/peripherals via AXI interfaces in Zynq-7000 AP SoC

- Provides L1 cache coherency between the Cortex-A9 processors and the Accelerator Coherence Port
- Arbitrates between Cortex-A9 processors requesting L2 cache accesses
- Provides access to the on-chip ROM and RAM
- Manages Accelerator Coherence Port (ACP) accesses.

1.3.1.4 Accelerating Software with Hardware
The Accelerator Coherence Port (ACP) has the following characteristics:

- The ACP is a slave AXI 64-bit port on the SCU connected to the PL
- ACP port can be used by accelerators in the PL to access the ARM L1/L2 caches and maintain coherency
- Caches accesses could be used as a mean to share data between the A9 CPUs and hardware accelerator in the PL with low latency

### 1.3.1.5 Cortex-A9 NEON Processing and Floating Point Unit

ARM based architecture uses a specific coprocessor that extends the ARM instruction set. The main features are:

- NEON SIMD (Single Instruction Multiple Data) engine accelerator for multimedia applications.
- Instructions need fewer cycles than ARM
  - NEON typically halves the number of processor cycles for a given application
- Enabled via software.

The Floating Point Unit (FPU) Engine is another coprocessor used as an extension to NEON sub-processor. The main features are:

- Hi performance single/double precision floating-point operations
- Register set shared with NEON
- 2 MFLOPS/MHz performance

### 1.3.2 Programmable Logic Overview

The PL is derived from Xilinx’s 7 series FPGA technology (Artix-7 for the 7z010/7z020 and Kintex-7 for the 7z030/7z045/7z100). The PL is used to extend the functionality to meet specific application requirements. The PL includes many different types of resources including configurable logic blocks (CLBs), port and width configurable block RAM (BRAM), DSP slices with 25 x 18 multiplier, 48-bit accumulator and pre-adder, a user configurable analog to digital convertor (XADC), clock management tiles (CMT), a configuration block with 256b AES for decryption and SHA for authentication, configurable Select I/O and optionally GTX multi-gigabit transceivers and integrated PCI Express (PCIe) block. For more information refer to [1].

### 1.4 Summary of the Next Chapters

Once some general questions of embedded systems and ZedBoard architecture have been introduced, it is time to introduce the goal of the next chapters.

Relying on Xilinx Design Tools and Open Source tools like BuildRoot among others, the first aim is to build our own embedded Linux system, in other words our own Linux distribution for ZedBoard, after that, a user space application which interact with the kernel space and some of the hardware of the Zynq-7000 SoC and ZedBoard is going to be built and debug.
The final result will be stored in a partitioned Secure Digital memory as depicted in Figure 3. The SD card memory has four files inside, the boot.bin, zImage, devicetree.dtb and ramdisk8M.image.gz. The following chapters detail how each block, depicted in aforesaid Figure 3, is built.

In Chapter 2: the configuration of the Processing System and the Programmable Logic will be described. This configuration relies on Xilinx Design Tools for setting up the basic parts of the processor and also the FPGA side. That configuration will be the first steps for building the red block BOOT.BIN of Figure 3, numbered 1. Also some general questions of the rest of the development tools are going to be introduced in Chapter 2:

Chapter 3: detailing the booting process since the power on and with the design project created in Chapter 2:, the aim of this chapter is to understand the boot paradigm of the device and build boot.bin file and the devicetree.dtb numbered 1 and 2 respectively in Figure 3. The last section briefly explain the road map for building a Linux OS from sources, the main files that permits to understand the structure and operation of the build system; leaving basic concepts clear for going into the detail of building an entire distribution with an automated open source build tool in the next chapters.

Chapter 4: will describe the set-up of an open source build platform for creating an entire Linux embedded distribution and setting a cross building platform for embedded Linux programs development. Finally zImage and ramdisk8M.image.gz numbered 3 and 4 respectively in Figure 3 will be generated.

Chapter 5: once the Linux distribution is running in our target, the ZedBoard; the Linux Ethernet driver of the Zynq-7000 has been modified for adding time-stamping functionalities for implementing the Precision Time Protocol IEEE1588 Standard for precision clock synchronization for networked measurement and control systems.

Chapter 6: Recapitulate the achieved objectives of this project and the future improvements for the system.
Chapter 2: Development Tools

2.1 Design Flow for Zynq-7000

Figure 4 shows the design flow for Zynq-7000 SoC, as introduced in the previous chapter, the aim is to use ZedBoard as Embedded Linux development Platform. The steps below describe the main building aspects that will be deeply explained from now on.

1. The design and implementation process begin with launching the PlanAhead tool, which is the central tool from the beginning till Bitstream generation is completed. Then the Hardware will be exported to SDK where application development takes place.
2. Including ARM Cortex A9 Processing System (PS) in the project will cause step 3.
3. XPS is automatically launched from PlanAhead for configuring the PS and the optional addition of Programmable Logic (PL) peripherals.
4. Processing System 7 wrapper (PS7) instantiates the Processing System section of the Zynq-7000 for the Programmable Logic and the external board logic.
5. PS7 settings have to be configured to select the ZedBoard, adding PS I/O peripherals, memory configurations, clock frequencies, etc.
6. Optionally IP Cores can be added or created. When finished, XPS have to be closed and return to PlanAhead. XPS will create `ps7_init.tcl, ps7_init.c, ps7_init.h` which contains the minimal peripheral configuration for the PS like clocks and memory controllers, `XML File` which contains the processor and peripheral instantiation and addresses for FSBL and BSP generation. And `MHS File`, the hardware netlist of the processor subsystem which fully defines the embedded system hardware.
7. When finished, close XPS to return to PlanAhead.
8. Back in the PlanAhead tool, a top-level HDL wrapper has to be generated for the processing system.
9. If the design need constraints, typically used to ensure that signals are routed properly, an `*.ucf` file has to be created in or imported to PlanAhead.
10. Generation of the Bitstream for configuring the logic in the PL if soft peripherals or other HDL are included in the design, or if hard peripheral input or output was routed through the PL. At this stage, the hardware has been defined in `system.xml`, and if necessary a `system.bit` Bitstream is generated. The Bitstream could be programmed into the FPGA or later from within SDK.
11. Now that the hardware portion of the embedded system has been built, it is going to be exported to the SDK to create the software design.
12. Software project associated with the hardware design has to be added. Within SDK, for operating system development, a First Stage Boot Loader (FSBL) application project and a Device
The Device Tree Support Package have to be created. The Device Tree generation will be detailed later.

13. The First Stage Boot-Loader will cause an executable file, `FSBL.elf`, compiled against Zynq-7000 ARM architecture.

14. The combination of the FSBL, the Second Stage Boot Loader (U-Boot) and the Bitstream in a Boot Image have to be loaded together with the Device Tree Blob, the Linux Kernel Image and the Filesystem for running the operating system.
The Zynq-7000 Processing System can be used independently of the Programmable Logic which eliminates step 8, step 9 and step 10. Logic in the PL can be designed to operate independently of the PS, however PS or JTAG must be used to program the PL.
A typical Bare-Metal design flow for Zynq-7000 will not differ much from what it is explained above. Not having the necessity of a Second Stage Boot-Loader, a Bare-Metal Board Support Package that host a collection of libraries and drivers forming the lowest layer of the application will be created in that case.

### 2.2 Xilinx Design Tools

#### 2.2.1 Plan Ahead

This tool is a design and analysis software that provides an environment for the FPGA design implementation process and act as a nexus of ISE design tools for building entire System on Chip applications. For the aim of this document, the main tools that are integrated in PlanAhead are:

- Xilinx Embedded Development Kit (EDK)
  - Xilinx Platform Studio (XPS)
  - Software Development Kit (SDK)
- ChipScope debugging tool
- iMPACT device programming tool

The Embedded Development Kit is a suite of tools and IP that can use to integrate the hardware and software system components. Xilinx Platform Studio can be used to design the hardware portion of the embedded processor system. Specification of the microprocessor, peripherals, and interconnection of these components, along with their respective detailed configuration takes place in XPS.

The recommended flow is to start with the PlanAhead tool to manage the project for the FPGA design and handling the embedded processor design as a single source file developed and managed within XPS.

#### 2.2.1.1 Main Features

Some examples when PlanAhead tool is used:

- Manage the design data flow from RTL development through Bitstream generation
- Manage constraints and perform Floorplanning
- Perform Design Rule Checks (DRC’s)
- Debug with ChipScope debugging tool
- Analyse implementation results.
- Perform I/O pin planning
- Configure and launch multiple synthesis and implementation runs
2.2.1.2 Creating a New Project

Launching PlanAhead and clicking Create New Project

A New Project window pop-up is displayed, as shown in Figure 6, and following the instructions the name and location of the project will be set.
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.

Several types of projects can be created, like Figure 7 shows. A Register Transfer Level (RTL) project is selected to manage the entire System on Chip design flow.

Not checking **Do not specify sources at this time** in Figure 7 will skip the steps of adding sources files and constraints and allow to select the target part and create the project.

![PlanAhead Creating A Project 3](image)

In the next step, see Figure 8, the target device is selected. This can be done by specifying a specific part or by selecting a development board. Selecting **Zynq-7000** on **Family** and **Sub-Family** sections, **Package clg484**, **Speed grade -1**, and **Temp grade C** should leave only one device in the **Device Table**.
A project summary is displayed, see Figure 9. Clicking Finish the project will be built and the main PlanAhead window is displayed as shown in Figure 10.
2.2.1.3 Adding Embedded Sources

The current project is blank. To access the ARM processing system, an Embedded Source has to be added to the PlanAhead project. Once the Embedded Source is created Xilinx Platform Studio will allow the developer to configure the embedded system. In the Flow Navigator pane click the Add Sources icon as displayed in Figure 11.

![Figure 10 PlanAhead Creating A Project 6, PA Main Window](image)

![Figure 11 PlanAhead Adding Embedded Sources 1](image)
The Add Sources wizard will open, **Add or Create Embedded Sources** option has to be selected, as shown in Figure 12, and click **Next**.

![Figure 12 PlanAhead Adding Embedded Sources 2](image)

In the Add or Create Embedded Source window, click **Create Sub-Design**, type a **Module Name** and click **OK**.

![Figure 13 PlanAhead Adding Embedded Sources 3](image)
A Xilinx Microprocessor Project (XMP) file is the top-level file descriptor of the embedded system. All project information is stored in the XMP file, XPS reads this file and graphically displays its contents on the XPS user interface. As shown in Figure 14, the Add Sources wizard creates a file named system.xmp which 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.

As depicted in Figure 15, when PlanAhead recognizes that the design includes an embedded processor, ISE automatically starts XPS and opens the Base System Builder to complete the design.
2.2.2 XPS

Xilinx Platform Studio is the development environment used for designing the hardware portion of the embedded processor system. Specification of the processor, peripheral and the interconnection of these components, along with their respective configuration, takes place in XPS.

2.2.2.1 Main Features

- Ability to add processor and peripheral cores, edit core parameters, and make bus and signal connections to generate a Microprocessor Hardware Specification (MHS) file.
- Ability to generate and view a system block diagram and/or design report.
- Project management support.
- Process and tool flow dependency management.
- Ability to export hardware specification files for import into SDK

2.2.2.2 The Xilinx Platform Studio Project

A new embedded system can be designed in XPS using two methods:

- Using the Base System Builder (BSB) wizard. Where the processing system I/O interface can be selected and default peripherals can be added.
- Creating a Blank Project. Within this option, the developer manually must add Processing System to the design and do the configuration for I/O interface.
A dialog box will open asking if the developer wants a Base System using BSB wizard to be created. Click No, then the main XPS window will appear empty, see Figure 16.

Since use Base System Builder is not chosen, the ARM Processing System must now be added manually, like Figure 17. In the IP Catalog→Processor double click Processing System.

Click Yes to confirm adding the PS7 IP to the design as shown in Figure 18.
The Processing System 7 IP is the software interface around the Zynq Processing System. As known, Zynq-7000 family consist of 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).

![Image of System Assembly View](image1)

In the **System Assembly View → Bus Interfaces** tab it is seen that `processing_system7` was added, see Figure 19.

![Image of XPS PS7 Bus Interfaces Tab](image2)
This Processing System is completely unconfigured, as indicated by all the I/O Peripherals being gray (non selected), see Figure 20. All PS features are in their default state, ready to be customized, which is what it is going to be made. The green coloured blocks in the Zynq Processing System diagram are items that are configurable. You can click a green block to open the corresponding configuration window.

Clicking anywhere on the peripherals opens the **Zynq PS MIO Configurations** dialog. The bank voltage decisions on ZedBoard were made based on the interfacing devices like QSPI, PHY’s, etc. The peripherals are listed from top to down in order of priority based on either importance in the system, like the Flash, or how limited they are in their possible MIO mapings. The least flexible are at the top, while the most flexible, GPIO, is at the bottom.

The configuration of the peripherals could be made one by one or could be based on a configuration file. The configuration file provided in the **Zedboard Community → Documentation → Board Definition Files** [2] can be loaded clicking **Import** in the **Zynq tab view**. In the second box click the **Plus sign button** to add the path of the configuration file, once finished click **Ok** as shown in Figure 21.
Click **Yes** in the confirmation window that opens to verify that the Zynq MIO Configuration and Design will be updated.

As shown in the Figure 22, many peripherals are now enabled in the Processing System with some MIO pins assigned to them in coordination with the board layout of the ZedBoard. For example, UART1 is enabled and UART0 is disabled. This is because UART1 is connected to the USB-UART bridge chip on this board.
The boot device is one of the most important parts. Zynq-7000 allows to select just one, QSPI, NOR or NAND. SD Card is also a boot option and will show up lower in the list. It can be seen in Figure 21 that Quad SPI Flash is selected and NOR and NAND peripherals are greyed out because the three interface are mutually exclusive. Other important part to notice is, in Figure 22, MIO 0 and from MIO 9-15 eight I/Os mapped to the PS Pmod.

When finished, close the Zynq PS MIO Configurations window and then the System Assembly View will appear and the new peripherals selected will be colored.

### 2.2.2.3 Set the PS PLL Clocks

As described in [3], “Zynq-7000 AP SoC’s PS subsystem uses a dedicated 33.3333MHz clock source IC18. The PS infrastructure can generate up to four PLL-based clocks for the PL system. An on-board 100MHz oscillator, IC17, supplies the PL subsystem clock input on bank 13, pin Y9”.

Clicking on the box for Clock Generation, the Clock Wizard will open. The Zynq-7000 PS has three PLL the ARM PLL, the DDR and the I/O PLL. Each uses the same input reference clock, which is 33,333MHz on ZedBoard.

Each PLL must be set to operate in a specific frequency range, as given by the datasheet. Note that for the -1 device, this range is 780 MHz to 1600 MHz as depicted on Figure 24.
Once the PLL output frequency is set, then to generate the desired clock limitations come by the integer dividers.

The CPU frequency, the DDR frequency and the ZedBoard input frequency command the PLL output frequency for the ARM and DDR PLLs, both frequencies must be multiples of 33.3333 MHz. For instance, the CPU frequency must be an integer divider of the ARM PLL, ARM PLL must be set to (33.333 MHz * 40) and the CPU divider must be 2.

Everything else that uses these PLLs must now use an integer divider of the set output frequency. Other point to focus is the Actual Frequency 190.476 MHz of the QSPI peripheral interface as shown in Figure 25. The QSPI peripheral interface operates at maximum rate of 100 MHz and has an internal divider of 2. So 190.476 MHz should make sense since there is no internal divider that can generate exactly 200 MHz from the ARM PLL’s 1333.333 MHz. Hence, one approximation to achieve that frequency could be dividing 1333.333/7.

The I/O PLL has also some prioritized dependencies, for our design the most dominant one is the Ethernet. Ethernet functionality is dependent on having an accurate clock. The user has to set the ENET0 to 1000 Mbps. This result is achieved by an integer multiplier of value equal to 30, so the operation 33.333 MHz * 30 will result the desired rate.
2.2.2.4 Configuration of the DDR Controller

ZedBoard includes two 32-bit DDR3 memories. The PS incorporates both the DDR controller and the associated Physical Layer Interface, including its own set of dedicated I/Os. DDR3 memory interface speeds up to 533 MHz. For the best DRR3 performance, in the PS Configuration Tool in Xilinx Platform Studio DRAM training is enabled.

The PS Configuration tools’ Memory Configuration Wizard contains two 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 in the ZedBoard PCB trace length reports provided by [2] ZedBoard’s community web page in documentation area.

The PS7 DDR Configuration screen, shown in Figure 26, allows for configuration of the DDR Controller, the Memory Part, and the board details used for DDR interface. Click to Enable DDR Controller and make sure that Memory Type is assigned to DDR 3. Clicking on Expand to calculation delay will open the area for entering PCB trace lengths as depicted in Figure 28. Notice that there are four byte groupings, each of which contains: clock CLK, strobe DQS and 8 bits of data DQ.
For getting the trace lengths the layout of the ZedBoard will have to be opened, but other option is to take the parameters measured by Xilinx contained in Table 2 DDR3 Worksheet Calculations of [3] ZedBoard Hardware’s User Guide, as shown in Figure 27.

<table>
<thead>
<tr>
<th>Pin Group</th>
<th>Length (mm)</th>
<th>Length (mils)</th>
<th>Package Length (mils)</th>
<th>Total Length (mils)</th>
<th>Propagation Delay (ps/inch)</th>
<th>Total Delay (ns)</th>
<th>DQS to CLK Delay (ns)</th>
<th>Board Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKO</td>
<td>55.77</td>
<td>2195.9</td>
<td>470</td>
<td>2665.9</td>
<td>180</td>
<td>0.427</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLK1</td>
<td>55.77</td>
<td>2195.9</td>
<td>470</td>
<td>2665.9</td>
<td>180</td>
<td>0.427</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLK2</td>
<td>41.43</td>
<td>1631.1</td>
<td>470</td>
<td>2101.1</td>
<td>180</td>
<td>0.336</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLK3</td>
<td>41.43</td>
<td>1631.1</td>
<td>470</td>
<td>2101.1</td>
<td>180</td>
<td>0.336</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DQS0</td>
<td>51.00</td>
<td>2008.0</td>
<td>504</td>
<td>2512.0</td>
<td>180</td>
<td>0.402</td>
<td>0.025</td>
<td></td>
</tr>
<tr>
<td>DQS1</td>
<td>50.77</td>
<td>1998.8</td>
<td>495</td>
<td>2493.8</td>
<td>180</td>
<td>0.399</td>
<td>0.028</td>
<td></td>
</tr>
<tr>
<td>DQS2</td>
<td>41.59</td>
<td>1637.6</td>
<td>520</td>
<td>2157.6</td>
<td>180</td>
<td>0.345</td>
<td>-0.009</td>
<td></td>
</tr>
<tr>
<td>DQS3</td>
<td>41.90</td>
<td>1649.4</td>
<td>525</td>
<td>2484.4</td>
<td>180</td>
<td>0.398</td>
<td>-0.061</td>
<td></td>
</tr>
<tr>
<td>DQS7</td>
<td>50.63</td>
<td>1993.3</td>
<td>465</td>
<td>2458.3</td>
<td>180</td>
<td>0.393</td>
<td>0.410</td>
<td></td>
</tr>
<tr>
<td>DQS15</td>
<td>50.71</td>
<td>1996.4</td>
<td>480</td>
<td>2476.4</td>
<td>180</td>
<td>0.396</td>
<td>0.411</td>
<td></td>
</tr>
<tr>
<td>DQS23</td>
<td>40.89</td>
<td>1609.9</td>
<td>550</td>
<td>2159.9</td>
<td>180</td>
<td>0.346</td>
<td>0.341</td>
<td></td>
</tr>
<tr>
<td>DQS31</td>
<td>40.58</td>
<td>1597.8</td>
<td>780</td>
<td>2377.8</td>
<td>180</td>
<td>0.380</td>
<td>0.358</td>
<td></td>
</tr>
</tbody>
</table>

Filling the lengths shown in column two of Figure 27 in the PS7 DDR Configuration window as shown in Figure 28 will cause XPS to adjust delay parameters to achieve the best operation.
When you exit the XPS tool, 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. Expanding the Embedded Design Sources in the Sources view displays the various target files associated with the sub-design.

XPS will create `ps7_init.tcl`, `ps7_init.c`, `ps7_init.h` which contains the basic peripheral configuration for the PS, like clocks and memory controllers, `XML` file which contains the processor and peripheral instantiation and addresses for FSBL and BSP generation. And MHS file, the hardware netlist of the processor subsystem which fully defines the embedded system hardware.

Targets are the different design elements of the XPS sub-design that are required to support the object on the current project. These include a top module definition, an Instantiation Template, the synthesized Netlist, and any supporting documents such as log and datasheets. The project file for the embedded design `system.xmp` is displayed in the Hierarchy tab of the Sources view.

### 2.2.3 Design Constraints

The tools have internal awareness of the PS pin location mapping and the timing, based on what has being entered in the PS Configuration Tool. Nevertheless, the PL needs a User Constraint File (UCF) to define the pin locations and PS timing, with the exception of circuitry driven from a PS fabric clock.

In the Flow Navigator, **Project Manager** click **Add Sources**, as shown in Figure 29, a wizard to Add Sources will pop up, **Add or Create Constraints** has to be selected, then click **Next**.
As shown in Figure 30, click **Add Files...** Select *zedboard_master_UCF_RevC_v3.ucf*, previously this file has to be downloaded from [2] Documentation area of Zedboard’s Community webpage, click **OK**. This will add some basic constraints written by [5]Avnet Inc. about pin location assignments.
2.2.4 Synthesis, Implementation and Bitstream Generation.

The hardware platform is now configured. The configuration includes clock and DDR controller settings, it also enables and maps a UART peripheral. Now the hardware platform has to be built and exported to the Software Development Kit (SDK) so that applications can be developed.

A top level wrapper for the design will be created. Expand Design Sources in the Sources pane right click system (system.xmp) and select Create Top HDL as depicted in Figure 31.

PlanAhead generates a system_stub.vhd top-level module for the design. Notice that the embedded system, system.xmp is now a sub-module of system-stub.

As the typical building process flow for FPGA designs well documented in books and all over Internet, the PlanAhead application includes all necessary tools, like map, place and route, synthesis, implementation and Bitstream generation tools. In addition to the functionalities mentioned, it allows multiple run attempts with different RTL source versions, constraints or different strategies for synthesis or implementation as detailed in [6]. Figure 32 shows a high level scheme of this process.
- **Design entry**: After creating a project and adding timing, placement, and pinout constraints as required, a behavioral simulation can be performed in the design before synthesis, typically to verify that the design is functioning as intended.
• **Design Synthesis**: The synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture selected. One of the notable files resulted is a Netlist NGC file generated by Xilinx Synthesis Technology (XST).

• **Design Implementation**: After Synthesis design implementation has to be run, which compromises the following steps:
  a. Translate process creates a Native Generic Database (NGD) with the input Netlist and design constraints. The NGD generated describes the logical design reduced to Xilinx primitives. And creates a report BLD file.
  b. Map, which fits the logic defined by the NGD file into FPGA elements (CLBs and OIBs). The output is a Native Circuit Description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA and a Physical Constraints File (PCF).
  c. Place and Route takes the mapped NCD file, places and routes the design and produces an NCD file that is used as input for Bitstream generation.
  d. Programming file generation creates a Bitstream file that can be downloaded to the device.

• **Functional Verification**: The functionality of the design can be verified at different points in the design flow:
  a. Before Synthesis: RTL simulation typically to verify that the design is functioning as intended.
  b. Performing a Post-Translate simulation to verify that the source of design is correct after the Translate process.
  c. After configuring the device the FPGA design can be debugged using ChipScope.

• **Timing Verification**:
  a. Post-map static timing analysis report lists the signal path delays in the design derived from the design logic. Useful for evaluating timing performance of the logic paths.
  b. Post-place and route static timing report incorporates timing delay information to provide a comprehensive timing summary of the design.
  c. Post-map Simulation on the design to verify that functionality is correct after the Map process
  d. Post-place and route Simulation, also known as timing simulation. This process uses the post-place and route simulation model and a Standard Delay Format (SDF) file for full timing analysis of block and net delays.

In the left pane of PlanAhead as shown in Figure 33 there are the main buttons to configure and launch these processes. Clicking in Generate Bitstream, even though no Synthesis or Implementation has been created, will launch automatically the entire process described above.
Now that the hardware portion of the embedded system has been built, it is going to be exported to the SDK tool to create the software design.

### 2.2.5 SDK

The Xilinx Software Development Kit (SDK) provides an environment for creating software platforms and applications targeted for Xilinx embedded processors. SDK works with hardware designs created with the Xilinx Platform Studio development tool. SDK is based on Eclipse open source standard.

Few concepts for understanding the tool are going to be explained briefly.

#### 2.2.5.1 SDK Basic Concepts: Workspaces

SDK uses the concept of workspaces to hold the software development work. A workspace is a directory in the file system that SDK uses to hold meta-information about the projects which users are working with. The workspace also contains SDK settings, software project files, and logs.

#### 2.2.5.2 SDK Basic Concepts: Board Support Packages

A software project contains one or more source files, along with the necessary head files, to allow compilation and generation of a binary output *.elf file.
A workspace can obtain multiple software projects. Each software project must have a corresponding board support package.

A board support package, BSP, is a collection of libraries and drivers that form the lowest layer of the application software stack. The software application must link against or run on top a given BSP.

### 2.2.5.3 SDK Basic Concepts: Hardware Platform

The Hardware Platform Specification file captures all the information and files from a Xilinx PlanAhead hardware design that is required for a software developer to write, debug and deploy software applications for that hardware.

Typically, someone who develops hardware using Xilinx Platform Studio (XPS) creates the Hardware Platform Specification in a directory. When developing software, this information is used in the Xilinx Software Development Kit (SKD). The main components of this specification are:

- A hardware description in an XML format file used to detect the processor and memory mapped peripherals present in the hardware.
- An FPGA Bitstream corresponding to the hardware description, used to program the Programmable Logic (PL) subsystem with the hardware created by the hardware designer.
- A series of source code files related to the Processing System (PS) initialization, `ps7_init.c`, `ps7_init.h`, `ps7_init.tcl` and `ps7_init.html`, which are directly related to the Zynq-7000 configuration in XPS.

### 2.2.5.4 Export Hardware to SDK

Software project associated with the hardware design has to be created within SDK. To follow the construction structure introduced in Chapter 1.4, Figure 34 shows the parts that the Software Development Kit will create.

As can be seen in Figure 34, one of the parts that compose `BOOT.BIN`, bootable image that prepares the system to support the Linux loader, is greyed out because U-Boot executable will be built separately and has to be imported to SDK for creating the bot image.
As shown in Figure 35, in PlanAhead top options bar, Click File → Export → Export Hardware for SDK.

As shown in Figure 36 a new window will pop-up for selecting the Source File to export and the location to ace the SDK files. Select Include Bitstream, Export Hardware and Launch SDK.
The PlanAhead design tool exported the Hardware Platform Specification for the design, `system.xml` to SDK. The `system.xml` file opens by default when SDK launches. The address map of the system read from this file is shown by default in the SDK window.

Four more files are exported to SDK, `ps7_init.c`, `ps7_init.h`, `ps7_init.tcl` and `ps7_init.html`. These files introduced above contain:

- 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
- The initialization data for
  - Processing System
  - PLLs
  - Clocks
  - DDR memory
  - MIO

SDK uses these settings 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 hardware platform is exported to SDK, Chapter 3: Linux on ZedBoard explains the building process using SDK and other external tools that will be introduced in this chapter.

### 2.2.6 Installing Device Tree Generator BSP

The Device Tree Generator is a Xilinx EDK tool that plugs into the Automatic BSP Generation features of the tool SDK. It will produce a Device Tree Source file that has the information of the hardware designed with PlanAhead and ISE design tools [7].
The Device Tree Generator is not included in the tools installed as default. It has to be downloaded from the Xilinx Github Repository with the following command.

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

Two files will be downloaded, `device-tree_vX.mld` and `device-tree_vX.tcl` merged in a file. For adding the Device Tree generation functionality in EDK a new folder has to be created in the path which are all the BSP generators, `<Xilinx_ise_installation_path>/ISE_DS/EDK/sw/lib/bsp/`, in the folder created the files downloaded have to be copied inside as shown in Figure 37.

![Figure 37 Adding Device Tree BSP Support in Xilinx EDK](image)

2.3 GNU Tools

2.3.1 Sourcery Codebench Lite

Xilinx provides some documentation and additional tools in an Open Source Wiki Internet page [8].
The GNU tools for Linux hosts are available for download as an installer in [9]. For launching the installer, a Linux console has to be opened and redirected to the folder containing the `xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin` file. Type the following command to launch the installer as shown in Figure 38.

```
$ sh xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin
```

![Figure 38 Sourcery Codebench Lite for Xilinx GNU/Linux Launching the Installer](image)

Accepting the terms of the License Agreement a list of the tools contained in the installer will be displayed like Figure 39.
After choosing the install set of tools and the installation folder, the \texttt{PATH} environment variable can be modified for adding the path of the binaries of the Sourcery CodeBench tool, this option can be chosen in the step shown in Figure 40.
The PATH environment variable would be set to 
<SourceryCodeBench_installation_path>/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_GNU_Linux/bin where the binaries are placed, as depicted in Figure 41.
This is used for building the Second Stage Boot-Loader U-Boot as explained in Chapter 3: Linux on ZedBoard.

### 2.3.2 BuildRoot
Extensively explained in Chapter 4: BuildRoot, the Build System. BuildRoot is a tool-chain that makes possible the generation of an entire embedded Linux system. It permits the generation and configuration of various types of Filesystems, the generation of a full cross-compiler tool-chain for building kernel images, user space programs and a large etcetera.

BuildRoot’s intention of use is to create an entire building platform of embedded Linux, device drivers and User Space programs development.

### 2.3.3 BusyBox
A Linux system needs a basic set of programs to work, like an init program, a shell, basic utilities for file manipulation and system configuration, etc.

BusyBox can be customized to provide a subset of over two hundred utilities in a single binary, each of which can be accessed by calling the single BusyBox binary with various names in a specific manner with the appropriate arguments. For each application integrated, a symbolic link to bin/busybox is created.

In normal Linux systems, those programs are provided by different projects that are not designed with embedded systems constraints in mind; therefore, BusyBox is an alternative solution extremely common on embedded systems.

Other very important job of BusyBox is to build an initrd Image, responsible for enabling early user space processing, as explained in Error! Reference source not found. Error! Reference source not found., initial RAM disk (or initrd) is a small root file system that usually contains directives to load specific device drivers before the completion of the boot cycle. When the kernel mounts the initial RAM disk, it looks for a specific file called linuxrc, treating this file as a script file executing commands contained inside. This mechanism is enabled for the system designer to specify the behaviour of initrd.

Download and configuration of BusyBox is integrated in BuildRoot as can be seen in 4.3 Buildroot Configuration.
2.3.4 Digilent Sources

Digilent provides a Git source repository that inherits from the Xilinx Linux Kernel project which includes support and drivers for many different Xilinx products. Digilent adds support for their system boards and Pmods and add-on boards.

2.3.4.1 U-Boot

U-Boot acts as Second Stage Boot Loader loading to RAM and executing the Linux Kernel Image. Digilent added support for ZedBoard in the U-Boot sources, implementing the Zynq-7000 boot mechanisms for loading the Kernel Image, the Device Tree file and the Root Filesystem Image. If U-Boot support for ZedBoard boot mechanisms had not been implemented it would have to be built based on other similar device.

For obtaining U-Boot sources, they have to be downloaded from the Digilent Git repository.

```
$ git clone git://github.com/Digilent/u-boot-digilent.git
```

2.3.4.2 Linux Kernel Sources

Xilinx Open Source Linux development project is a customized solution based on 3.X Linux Kernel from kernel.org, not a commercial solution. Intended to run on Zynq-7000 platform (ARM) among others it includes specific configurations, board support packages and drivers specific to this platform. Digilent adds extra configurations and drivers for Zedboard for example Digilent Peripheral Modules (Pmod) interfaces.

The Digilent Linux kernel sources will be downloaded from the Git repository in the BuildRoot building process. But also can be downloaded with the following command.

```
$ git clone git://github.com/Digilent/linux-digilent.git
```
Chapter 3: Linux on ZedBoard

3.1 Overview

First, it is going to be described how configuration and boot occurs on ZedBoard, divided in four stages of non-secure boot. The first one, non-user configurable, called Stage 0, is the one who loads executable code of First Stage Boot Loader then the FSBL takes control and prepares the system so a larger boot loader can be loaded. Stage 2 or U-boot Boot-Loader locates, load and execute the Linux kernel, also passes the Device Tree to the kernel. Then the Operative System initialize system hardware and mount root file system. A diagram of this process can be seen in Figure 42.

3.2 Stage 0: BootROM

An internal BootROM stores the stage-0 boot code, which configures one of the ARM processors and the necessary peripherals to start fetching the FSBL boot code from one of the boot devices. The programmable logic is not configured by the BootROM.
ROM code detects desired boot mode (NAND, QSPI, JTAG) and loads executable code of FSBL from selected peripheral interface.

PS configuration starts after power-on reset. The ARM CPU starts executing code from the on-chip BootROM with JTAG disabled. The BootROM contains code for base drivers for NAND, NOR, Quad-SPI, SD and PCAP. DDR and other peripheral initializations are nor performed, and must be done in the stage 1 image or later.

The PS boot source is selected via the mode-pin signals indicated by a weak pull-up or pull-down applied to specific pins as can be seen in Figure 43 extracted from [1].

The BootROM supports encrypted and unencrypted images referred to as secure boot and non-secure boot, respectively. In secure boot the CPU, running from secure BootROM code, decrypts and authenticates the incoming user PS image, stores it in the OCM RAM, and then branches into it. In non-secure boot, the user can proceed with either secure or non-secure subsequent boot stages. Following a non-secure first stage boot, only non-secure subsequent stage boots are possible.

There are five possible boot sources: NAND, NOR, SD, Quad-SPI and JTAG. The first four sources are used in master boot methods in which the CPU loads the external boot image from non-volatile memory into the processing system (PS).

Among the possibilities enumerated above, the one used for ZedBoard is a non-secure master boot, where MIO[5:3] pins are set to 110, mode value stored to enable boot from Secure Digital Card. Performing these steps while booting:

1. Initializes de MIO pins SDIOCLK, SDIOCMD and SDIODATAIO[0:3].
2. Sets the SD clock to run at 226kHz with a 33.333MHz PS_CLK.
3. Reads BOOT.BIN from the root of the SD file system and copies it into OCM parsing the required BootROM header.
4. Starts execution from the beginning of OCM.
Then BootROM uses the Device Configuration Interface DMA to load the FSBL into on-chip RAM and releasing control to FSBL BootROM shuts down.

### 3.3 Stage 1: First Stage Boot-Loader

The First Stage Boot Loader is responsible for:

- Initialization using the PS configuration data provided by XPS.
- Programming the PL using a Bitstream (optional).
- Loading the second stage boot-Loader or bare-metal application code into memory.
- Starting execution of the second stage boot-loader or bare-metal application.

Using the Zynq-7000 configuration user-interface, XPS generates code from initialization of the MIO and SLCR registers. First stage boot-loader is generated by SDK.

The Bitstream for the PL and the second stage boot-loader or bare-metal application data, as well as other code and data used by the second stage boot-loader, Linux, or bare-metal application can be grouped into partitions in the flash image. The FSBL goes across the partition header to find the Bitstream and second stage boot-loader or bare-metal application partition. Using the Bootgen program with a wizard in the SDK, the FSBL is wrapped with the Bitstream (optional) and the bare-metal application or the second stage boot-loader in a single `BOOT.BIN` file.

### 3.3.1 Building FSBL

As explained in Chapter 2: section 2.2.5.4 Export Hardware to SDK, PlanAhead exports the hardware design and the Bitstream then SDK opens automatically. In the Project Explorer, left part of the window the System Hardware Platform folder contains the files needed for building the First Stage Boot-Loader as can be seen in Figure 44 which shows a high-level diagram of the process.
Once SDK has launched, the `system.xml` file placed in the Project Explorer pane, should be open in the main editor’s pane as shown in Figure 45. The `system.xml` contains the memory map and associated IP block for each of the hardware peripherals that were connected to the processing system in the XPS.
The `system_hw_platform` project also contains a Bitstream file, `system.bit`, as well as `ps7_init.c`, `ps7_init.h`, `ps7_init.tcl` and `ps7_init.html` files described in previous chapters. These files contain information of critical Processing System initialization and were automatically generated based upon the XPS Processing System settings during Hardware Export process.

SDK is now ready for software development. Clicking on **File → New → Application Project** like shown in Figure 46 the application project wizard will pop up as depicted in Figure 47.
A project name has to be written, then going to the **Next** step, the default templates appear, among them, **Zynq FSBL** is going to be selected, see Figure 48.
The FSBL and BSP projects are now created within the workspace and automatically built by the compiler. When the built process finishes the aspect of SDK will be as depicted in Figure 49.
Exploring the generated files using the Project Explorer pane to the left of the screen, if the file named `lscript.ld` is opened, the application linker script shows the mapping of two memories, see Figure 50, the `ps7_ram_0_S_AXI_BASEADDR` is the ARM local OCM memory location following a reset. The second memory segment listed as `ps7_ram_1_S_AXI_BASEADDR` is other partition of the same memory. The primary purpose of a linker script is to describe the memory layout of the target machine, and specify where each section of the program should be placed in memory. A Linker Script can be built manually with right-click in the application folder → Generate Linker Script.
3.4 Stage 2: Second Stage Boot-Loader, U-Boot

Now that the `zynq_fsbl.elf` has been built, it is time to come into Stage 2 building the Second Stage Boot-Loader. It will be finally placed in the boot medium as the orientate Figure 51.
The U-Boot Universal Boot-Loader is a GPL cross-platform boot loader pioneered by project leader Wolfgang Denk and forged by developer 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. So it is a good solution for Zynq Second Stage Boot-Loader.

U-Boot can obtain a kernel image from a SD Card, partitioned QSPI Flash, and even through Ethernet using TFTP (having a functional TFTP server). By default, U-Boot starts the procedure called `autoboot`, which looks for `BootMode` pins settings again for the source of the kernel image, the Device Tree Blob and the File System image.

### 3.4.1 Building the U-boot Bootloader

The U-Boot sources used for this system are not the official sources placed in the DENX git repository [10], as mentioned above Digilent has its own modified sources for supporting their hardware. So first of all, it is required to download the Digilent U-Boot sources from git.

```
$ git clone git://github.com/Digilent/u-boot-digilent.git
```

For a boot-loader to be useful across many processors and architectures, some method of configuring the boot-loader is necessary. As with a Linux kernel itself, a boot-loader is configured at compile time. Board-specific configuration is driven by a single header file specific to the target platform, together with the correct subdirectories based in the target board, architecture and CPU.
For this element of the boot process a Cross Compiler, from Sourcery Codebench and provided by Xilinx, is going to be used. Before starting to build the sources, the configuration of the cross-compiling toolchain must be done.

1. Linux environment variable `CROSS_COMPILE` has to be set to `arm-xilinx-linux-gnueabi-`
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, a scheme of the build process can be seen in Figure 52.

For configuring U-Boot for one of its supported platforms can be done with the following command.

```
$ make <platform>_config
```

Where `<platform>` is one of the platforms supported by U-Boot. This platform configuration targets are listed in the top-level U-Boot `Makefile`. The following command is used for the default configuration of ZedBoard.

```
$ make zynq_zed_config
```

This command 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, typing `make` command, u-boot for Zynq’s ARM architecture will be built.

```
$ make all
```

Then the source folder as shown in Figure 53, will contain the target executable file needed to execute on Zynq.

![Figure 53 U-Boot Source File After 'make all'.](image-url)
Occasionally some edition of the configuration file for this board has to be done; in this case, it is considered that no edition will be made, because the default configuration is sufficient for our needs. But for further use, the mechanism is going to be explained briefly.

U-Boot is configured using configuration variables defined in a board-specific header file. Configuration variables have two forms, configuration options and configuration settings. The first ones are selected using macros in the form of `CONFIG_XXX`. Configuration settings are selected using macros in the form of `CONFIG_SYS_XXXX`. U-Boot configuration is driven by a header file dedicated to that specific platform that contains configuration options and settings appropriate for the platform. The source tree includes a directory where these board-specific configuration header files reside. They can be found in `u_boot_source/include/configs`.

Numerous of features and modes of operation can be selected by adding definitions to the board-configuration file. Listing below, the configuration header file for the ZedBoard.

Listing 1 zynq_zed.h

```c
/*
 (C) Copyright 2012 Xilinx
 * Copyright (c) 2012 Digilent. All right reserved.
 * Author: Tinghui WANG <steven.wang@digilentinc.com>
 *
 * Configuration for Zynq Evaluation and Development Board - ZedBoard
 * See zynq_common.h for Zynq common configs
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#ifndef __CONFIG_ZYNQ_ZED_H
#define __CONFIG_ZYNQ_ZED_H

#include <configs/zynq_common.h>

/* High Level Configuration Options */
#define CONFIG_ZED /* Community Board */

/* Default environment */
#define CONFIG_IPADDR 192.168.1.10
#define CONFIG_SERVERIP 192.168.1.50

#undef CONFIG_ZYNQ_XIL_LQSPI

/* No NOR Flash available on ZedBoard */
#define CONFIG_SYS_NO_FLASH
#define CONFIG_ENV_IS_NOWHERE

#undef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS
  "ethaddr=00:0a:35:00:01:22\" \
  "kernel_size=0x140000\" \
  "ramdisk_size=0x200000\" \
  "qspiboot=sf probe 0 0 0;\" \
  "sf read 0x8000 0x100000 0x2c0000;\"

```

Linux on ZedBoard
Development of Embedded Linux Applications Using ZedBoard

```
"sf read 0x1000000 0x3c0000 0x4000000;"
"go 0x8000\0"
"sdboot_linaro=echo Copying Linux from SD to RAM...;"
"mmcinfo;"
"fatload mmc 0 0x8000 zImage;"
"fatload mmc 0 0x1000000 devicetree_linaro.dtb;"
"go 0x8000\0"
"sdboot=echo Copying Linux from SD to RAM...;"
"mmcinfo;"
"fatload mmc 0 0x8000 zImage;"
"fatload mmc 0 0x1000000 devicetree.dtb;"
"fatload mmc 0 0x800000 ramdisk8M.image.gz;"
"go 0x8000\0"
"jtagboot=echo TFTPing Linux to RAM...;"
"tftp 0x8000 zImage;"
"tftp 0x1000000 devicetree.dtb;"
"tftp 0x800000 ramdisk8M.image.gz;"
"go 0x8000\0"
```

#include <config_cmd_default.h>
#define CONFIG_CMD_DATE /* RTC? */
#define CONFIG_CMD_PING /* Might be useful for debugging */
#define CONFIG_CMD_SAVEENV /* Command to save ENV to Flash */
#define CONFIG_REGINFO /* Again, debugging */
#undef CONFIG_CMD_SETGETDCR /* README says 4xx only */
#define CONFIG_TIMESTAMP /* print image timestamp on bootm, etc */
#define CONFIG_PANIC_HANG /* For development/debugging */

#define CONFIG_AUTO_COMPLETE
#define CONFIG_CMDLINE_EDITING

#undef CONFIG_SYS_PROMPT
#define CONFIG_SYS_PROMPT "zed-boot>"

/* this is to initialize GEM at uboot start */
#undef CONFIG_ZYNQ_INIT_GEM
/* this is to set ipaddr, ethaddr and serverip env variables. */
#define CONFIG_ZYNQ_IP_ENV

/* HW to use */
#define CONFIG_UART1
#define CONFIG_TTC0
#define CONFIG_GEM0
#define CONFIG_ZYNQ_GEM
#define CONFIG_XGMAC_PHY_ADDR 0

/* Physical Memory map */
#define PHYS_SDRAM_1_SIZE (512 * 1024 * 1024)

/* SPI Settings */
#define CONFIG_ZYNQ_SPI
#define CONFIG_CMD_SPI
#define CONFIG_SF_DEFAULT_SPEED 30000000
#define CONFIG_SPI_FLASH
#define CONFIG_CMD_SF
#define CONFIG_SPI_FLASH_SPANSION

/* Place a Xilinx Boot ROM header in u-boot image? */
#undef CONFIG_ZYNQ_XILINX_FLASH_HEADER

#ifdef CONFIG_ZYNQ_XILINX_FLASH_HEADER
/* Address Xilinx boot rom should use to launch u-boot */
#undef CONFIG_ZYNQ_XIL_LQSPI
#define CONFIG_ZYNQ_XIP_START XPSS_QSPI_LIN_BASEADDR
Here it is shown how the boot process is done. `sdboot` is the most usual way of bringing up this ZedBoard system, `mmcinfo` is used to initialize the SD Card, then fetches inside that memory can be done.

Using the multi-file image format for U-Boot, it can load and boot files from a FAT or VFAT system. The procedure `sdboot` does three things.

1. First, `sdboot` reads the kernel image `zImage` from the FAT partition and copies it to 0x8000: `fatload mmc 0 0x8000 zImage`.
2. `sdboot` reads the DTB file and loads to 0x1000000: `fatload mmc 0 0x1000000 devicetree.dtb`.
3. `sdboot` reads the compressed ramdisk file system image named `ramdisk8M.image.gz` and loads it to 0x800000: `fatload mmc 0 0x800000 ramdisk8M.image.gz`. After loading these files, U-Boot begins the execution at RAM address 0x8000 where `zImage` is located.

### 3.5 The Boot Image
Using the **FSBL** created before, the **hardware bitstream** file and the **U-Boot ELF** file that was built on the cross build platform, the boot image will be created using SDK. This will generate a file from which placed in the SD card as depicted in Figure 54, the ZedBoard can be booted into U-Boot.

Zynq-7000 BootROM is capable of booting the processor from several different non-volatile memory types, but it requires a data structure referred to as the Boot Image Format File (BIF) for instructions on how to parse the different boot components. For the most part, simple Linux systems require only three components within the boot image:

1. FSBL (Stage 1 boot loader)
2. Programmable Logic Hardware Bitstream
3. U-Boot (Stage 2 boot loader)

The BIF file specifies each component of the boot image, in order of boot, and allows optional attributes to be applied to each image component. Each image component is usually mapped to a partition, but in some cases, an image component can be mapped to more than one partition if the image component is not contiguous in memory.

Bootgen is a standalone tool for creating a bootable image suitable for Zynq-7000. The program assembles the boot image by prefixing a header block to a list of partitions. Optionally, it is possible to encrypt and authenticate each partition. The output is a single file that can be directly into a boot flash memory of the system. The tool is actually integrated into SDK for automatic image generation, or used in a command-line orientated script.
3.5.1 Building the Boot Image

Once SDK is open, the next step is to launch the Create Zynq Boot Image tool using Xilinx Tools→Create Boot Image menu item. As shown in Figure 55, once the boot image is created, the first file could be loaded into the SD card and tested.

![Diagram of boot image construction](image)

**Figure 55 Boot.bin Construction, Design Flow Detail**

In the top menu bar of the SDK tool click in Xilinx Tools→Create Boot Image as shown in Figure 56. Then, a new window will pop-up for selecting the files to add.
A new Boot Image Format (BIF) file must be created. In the Bif file drop down menu, select the Create a new bif file... option, depicted in Figure 57.

The order for the files placed into a boot image is important. First, the FSBL “elf” file, then the hardware Bitstream and last the u-boot “elf” file by clicking the Add button and selecting the file below.
Click the **Open** button to add the file to the list of partitions in the boot image. The final result of adding the files is shown in Figure 58.

![Create Zynq Boot Image](image)

**Figure 58 Selecting Appropriate Boot Image Partitions.**

Then Create Image button will launch Bootgen in the background as shown in Figure 59, tool explained above for constructing boot images for Zynq-7000 AP SoC.

Bootgen merges the BIT and the ELF files into a single boot image with the format defined in the Boot Image Format (BIF) file to be loaded into Zynq devices at boot time. Bootgen is called against the **bootimage.bif** file, which contains the format which defines which files are integrated and what order they are added to the binary output file. The BIF file is automatically created by the Zynq Boot Image tool prior to the bootgen operation.

![Bootgen Command](image)

**Figure 59 Bootgen Running in Background.**

Bootgen will create a **BOOT.BIN** file in the folder specified by **Output Folder**. The BootROM on Zynq-7000 will only be able to locate image file on the SD Card if it is named **BOOT.BIN**. Output file is
copied to the SD Card and booting the ZedBoard, if any key is pressed before the expiration of the countdown, the U-Boot prompt will appear.

### 3.6 The Device Tree

![Figure 60 Focus on the Device Tree](image)

The Linux Kernel is a piece of embedded standalone software running on hardware. The kernel provides a standardized interface for application programmers to utilize all hardware resources without knowing the details. Thus, the kernel has to know every detail about the hardware it is working on. The Linux Kernel uses the data structure known as Device Tree Blob or Device Tree Binary, to describe the hardware.

The DTB is a database that represents the hardware components on a given board and has been chosen as the default mechanism to pass low-level hardware information from the bootloader to the kernel. Similar to U-Boot or other low-level firmware, mastering the DTB requires complete knowledge of the underlying hardware.

The DTB is provided as a courtesy by the board/architecture developers as part of the Linux kernel source tree, but if some custom hardware is done, the DTB must be customized too.

#### 3.6.1 Device Tree Source

The device tree blob is compiled to generate the binary in the proper form for U-Boot and Linux to understand. The `dtc` compiler is provided with the kernel sources.
Some of the data shown in Listing 2 is self-explanatory. The flat device tree is made up of device nodes. A device node is an entry in the device tree, usually describing a single device or bus. Each node contains a set of properties that describe it. For more information go to [11]

Listing 2 ZedBoard DTS Extract

```
/dts-v1/
/
{
    model = "Xilinx Zynq ZED";
    compatible = "xlnx,zynq-zed";
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    interrupt-parent = <0x1>;
    memory@0 {
        device_type = "memory";
        reg = <0x0 0x20000000>;
    };
    chosen {
        bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw
earlyprintk rootfstype=ext4 rootwait devtmpfs.mount=0";
        linux,stdout-path = "/axi@0/uart@E0001000";
    };
    ...
    cpus {
        #address-cells = <0x1>;
        #cpus = <0x2>;
        #size-cells = <0x0>;
        cpu@0 {
            clock-frequency = <0x27bc86bf>;
            compatible = "xlnx,ps7-cortexa9-1.00.a";
            d-cache-line-size = <0x20>;
            d-cache-size = <0x8000>;
            device_type = "cpu";
            t-cache-line-size = <0x20>;
            t-cache-size = <0x8000>;
            ...
        };
        cpu@1 {
            clock-frequency = <0x27bc86bf>;
            compatible = "xlnx,ps7-cortexa9-1.00.a";
            ...
        };
    };
    axi@0 {
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
        ranges;
        uart@e0001000 {
            compatible = "xlnx,ps7-uart-1.00.a";
            reg = <0xe0001000 0x1000>;
            interrupts = <0x0 0x32 0x0>;
            interrupt-parent = <0x1>;
            clock = <0xe02fa080>;
        };
        eth@e000b000 {
            compatible = "xlnx,ps7-ethernet-1.00.a";
            reg = <0xe000b000 0x1000>;
            ...
        };
    }
```
In the device tree source file, the slash character “/” stands for the root node and everything inside the brackets “{ }” are either properties of the root node or the children nodes. In Listing 2, the first property of the root node is `model`. String “Xilinx Zynq ZED” is assigned to it. Property `compatible` defines the compatibility of the node, and, in this case, is given the compatibility string “xlnx,zynq-zed”.

### 3.6.2 Device Tree Generator

The goal of this section is building the project to obtain a *.dts file that can be compiled to generate a *.dtb file and placed in the SD Card as depicted in Figure 61, so then U-Boot can pass it to the Linux kernel.
The device tree generator is a Xilinx EDK tool that plugs into the Automatic BSP Generation features of the tool XPS. The purpose of the device tree generator is to produce a device tree source that has the information of the hardware design in the EDK project.

Once the design has been exported from Plan Ahead to SDK, previous configuration of the XPS being done, the first thing to do is to generate a Board Support Package.

First, in Plan Ahead the hardware has to be exported, File→Export→Export Hardware for SDK... exporting the Bitstream file and the system hardware platform required for the generation of the *.dts, as shown in Figure 62.
When the SDK finished launching, it is time to create the BSP, **File→ New→ Board Support Package** like Figure 63 shows.

![Figure 63 New Board Support Package For Device Tree Generator.](image)

Then, a new wizard appears to select what short of BSP the user wants, Figure 64.
Setting the path of the project where the user wants to allocate the BSP and clicking **Finish**, a new dialog appears, Figure 65, for configuring the **Bootargs**, for now it is going to be left in blank.
Clicking OK adds the device-tree-BSP to the SDK workspace and displays it in the Project Explorer, as shown in Figure 66. After SDK finish rebuilding the project, the new xilinx.dts file is created and located in <workspace>/<device-tree-bsp>/<processor-name>/libsrc/device-tree_v0_00_x folder. For now, bootargs will be left in blank.

**Figure 66 Hardware’s Device Tree Source generated.**

### 3.6.3 Device Tree Compilation

The DTS file needs to be compiled into a DTB file that kernel can understand. The device tree compiler (DTC), located under `scripts/dtc` in the Linux kernel source, will compile the DTS file into a DTB file with the following command.

```bash
$ ./scripts/dtc/dtc -I dts -O dtb -o <output_file_path>/<output_file_name>.dtb
<input_file_path>/<input_file_name>.dts
```

The DTC compiler can also de-compile a DTB file back to the DTS file with the command:
3.6.4 Linux Bootargs

Passing command line arguments to the Linux kernel allows for very flexible and efficient configuration, which is especially important in Embedded Systems. These features are nearly undocumented. One reason could be that they rely on the capabilities of the boot loaders.

U-Boot supports bootargs environment variable. Their contents are automatically passed to the Linux kernel as command line arguments. The advantage of this mechanism is that when the system finished starting, it would have some configurations already set.

There are other possibilities of passing some configurations to the kernel at boot time. The second way is setting the default kernel booting arguments in the kernel configuration menu at Boot Options -> Default kernel command string (CONFIG_CMDLINE).

The third way is filling the chosen node of the Device Tree as shown in Listing 2 ZedBoard DTS Extract, which will overwrite the default kernel booting arguments.

Listing 3 Bootargs in chosen node

```
chosen {
    bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M earlyprintk rootfstype=ext2 rootwait devtmpfs.mount=0; linux,stdout-path="/axi0/uart@E0001000";";
};
```

The meaning of these boot arguments shown in Listing 3 Bootargs in chosen node are explained in the next table.

Table 1 Devicetree Bootargs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console</td>
<td>ttyPS0</td>
<td>Default console</td>
</tr>
<tr>
<td></td>
<td>115200</td>
<td>ttyPS0 is the UART0 of the Zynq PS</td>
</tr>
<tr>
<td>Root</td>
<td>/dev/ram</td>
<td>RAM disk mounted as root</td>
</tr>
<tr>
<td></td>
<td>rw</td>
<td>Read and write privileges</td>
</tr>
<tr>
<td>initrd</td>
<td></td>
<td>Initial ram disk</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>earlyprintk</td>
<td></td>
<td>Enables kernel log messages that precede the initialization of the traditional console to be shown.</td>
</tr>
<tr>
<td>rootfstype</td>
<td>Ext2</td>
<td>Filesystem format</td>
</tr>
<tr>
<td>rootwait</td>
<td></td>
<td>The systems wait indefinitely for root device to show up</td>
</tr>
<tr>
<td>Devtmpfs.mount=0</td>
<td></td>
<td>This will instruct the kernel to automatically mount the devtmpfs filesystem at /dev, directly after the kernel has mounted the root filesystem</td>
</tr>
<tr>
<td>linux,stdout-path=</td>
<td>&quot;/axi@0/uart@E0001000&quot;</td>
<td>Routes the Kernel console to the standard input output UART0</td>
</tr>
</tbody>
</table>

For detailed information about the kernel parameters refer to kernel-parameters.txt under Documentation in the Linux kernel source. [12], Xilibus documentation about Zynq-7000 device trees [13] and The Denx U-Boot and Linux guide in the section about boot arguments [14].

The Linux kernel contains a mechanism to mount an early root file system to perform certain startup-related system initialization and configuration. This mechanism is known as initial RAM disk or initrd. The bootloader gets involved loading a compressed kernel image into the memory and then loads an initrd image into another section of available memory. As shown in Table 1 the initrd image has to be loaded and mounted from the physical memory location 0x800000 which has a size of 8MB.

### 3.7 Linux Kernel Basics

The best way to begin this section is referring to the very first lines of every Linux Kernel top source level README file.

#### Listing 4 Top-level Source README File for every Linux Kernel

```markdown
What is LINUX?

Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.

It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and multistack networking including IPv4 and IPv6.

It is distributed under the GNU General Public License - see the
```
The official Linux Kernel sources are placed in The Linux Kernel Archives [15], but there are a lot of other source trees outside kernel.org, most of them for specific development, that is the case of Linux Digilent [16] and Linux from Xilinx [17]. The changes applied to the sources that developers done, have to be accepted by the top maintainers.

The top-level source directory of the Linux kernel source tree contains the following subdirectories. The non-directories and the directories used for source control have been omitted.

- **arch**: The arch subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture, i.e. arm, microblaze, x86
- **include**: The include subdirectory contains most of the include files needed to build the kernel code. It has further directories including one for every architecture
- **init**: This directory contains the initialization code for the kernel
- **mm**: This directory contains all of the memory management code. The architecture specific memory management code resides in arch/<target_architecture>/mm/
- **drivers**: All of the system’s device drivers reside in this directory. They are subdivided into classes
- **ipc**: This directory contains the kernel inter-process communications code.
- **modules**: This is a directory used to hold built modules.
- **fs**: All of the file system code resides inside fs, sub-divided into directories one per supported file system.
- **Kernel**: The main kernel code.
- **net**: the kernel’s networking code.
- **lib**: This directory contains the kernel’s library code. The architecture specific library code can be found in arch/<target_architecture>/lib/
- **scripts**: This directory contains the scripts that are used when the kernel is configured.

Many of these subdirectories contain several additional levels of subdirectories containing source code, makefiles and configuration files. By far, the largest branch of the Linux kernel source tree is found under /drivers.

Additional files found in the top-level Linux subdirectory include the top-level makefile, a hidden configuration file and various files not involved in the build itself.
3.7.1 Linux Kernel Configuration and Build system

Linux kernel 3.2 contains approximately 14998651 lines of code

Before compiling the Linux kernel tree, it has to be configured for the architecture.

$ make ARCH=arm CROSS_COMPILE=arm-linux- digilent_zed_defconfig

The previous 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_COMPILE=arm-linux-.

For adding or subtracting extra configuration to the Linux kernel some configuration editors can be invoked.

- config: Update current config using a line-oriented program
- menuconfig: Update current config using a menu-based program
- xconfig: Update current config using a QT-based front end
- gconfig: Update config using a GTK-based front end

By typing make config, make menuconfig, make xconfig or make gconfig; a form of configuration editor is launched. When the configuration editor is exited the changes have to be saved, the .config file will be updated with the current configuration.

The .config file drives the kernel build via the top-level makefile. During the build process, the .config file is processed into a C header file found in the /include/linux directory called autoconfig.h; that is an automatically generated file.

A file called Kconfig exists in almost every subdirectory of the Linux kernel sources; Kconfig drives the configuration process for the features contained within its subdirectory. The contents of Kconfig are parsed by the configuration subsystem. The configuration utility like menuconfig or xconfig reads the Kconfig file stating from the arch/ subdirectory’s Kconfig file. Depending on which architecture you are building, the configuration utility reads this architecture-specific Kconfig as the top-level configuration definition.

Directives inside Kconfig file tells the configurator to read in another Kconfig file of another location, these Kconfig relations determine the complete set of menu options presented to the user when configuring the kernel.
When building the kernel, the Makefiles scan the configuration and decide what subdirectories to
descend into and what source files to compile for a given configuration.

The most common use of make is to specify no target, this generates the kernel *.elf file
vmlinux which is the default binary image for the chosen architecture. Many architectures and machine
types require binary targets specific to the architecture and the boot-loader used. One of the common
architecture-specific targets is zImage, the one used in this system. U-Boot will decompress it and load it
into the system.

For more information refer to [18].
Chapter 4: BuildRoot, the Build System.

4.1 Why using an embedded Linux Build Platform?

Building a custom embedded Linux distribution involves to create a kernel image with its required libraries; the packets for supporting hardware devices and software applications; the dependencies among packages ought to use for the user applications; the boot-loader with its required libraries; determine which versions of packages and tools are compatible with one another; build a file system; build the software applications that will run in the system and an extremely large etcetera of things to do. So, if anyone can be so courageous of building all without help, that person will waste a lot of precious time.

BuildRoot is a bunch of Makefiles and patches designed to build a complete embedded Linux distribution, it automates the building process of the embedded system.

- It can build all the required components for the embedded system, cross-compiling toolchain, root filesystem generation, kernel image compilation and boot-loader compilation.
- It allows configuration using menuconfig, gconfig and xconfig configuration interfaces.
- Supports hundreds of packages for userspace applications and libraries.
- Supports multiple filesystem types for the root filesystem image.
- Can generate an uClibc cross-compilation toolchain.
- Simple structure relying on Makefile language.

4.2 BuildRoot installation

4.2.1 System requirements

Buildroot is designed to run on Linux systems. It needs some software to be installed on the host system as: which, sed, make, binutils, build-essential, gcc, g++, bash, gawk, bison, flex, gettext, patch, gzip, bzip2, perl, unzip… The most of the packages are included in a desktop linux distribution.

For configuration interface dependences ncurses5 for menuconfig, qt4 for xconfig and glib2, gtk and glade2 for gconfig interfaces are needed.

For more information about host software requirements BuildRoot User Manual can be consulted [19].
4.2.2 Getting BuildRoot

The first thing to do is to locate and download a BuildRoot snapshot. For this guide release 2012_11_1 is the chosen one [20]. Other way is downloading using git.
4.3 Buildroot Configuration

Figure 67 BuildRoot Design Flow Overview
Figure 67 shows a basic diagram of the BuildRoot construction flow to achieve the generation of the remaining parts to build, the zImage and the Filesystem. Before this, some configuration of Buildroot tools have to be done.

4.3.1 Cross compilation tool-chain

A compilation tool-chain is a set of tools that allows the developer to compile code for the system. It consists of:

- A compiler, in our case gcc
- Binary utils like assembler and linker, in our case binutils
- A C standard library, in our case uClibc

The system installed on the developer's workstation, host machine, already has a compilation toolchain used to compile applications that runs on the system and will generate binary executables. After the user writes an application, he has to invoke a compiler that came with his desktop system to generate a binary executable image. That image is properly formatted to execute on the machine on which it was compiled, the host machine. This compilation toolchain is called native compilation. For embedded system development, it is usually impossible or not interesting to use a native toolchain, some reasons can be that the target is too restricted in terms of storage and/or memory, that the user may not want to install all development tools on the target or that the target is too slow compared to the workstation.

In a cross compilation environment, the compiler running on the host machine will generate binaries, which are incompatible with itself. They will only run on the target machine.

As shown in Figure 68, the workstation (x86) has the native tool-chain and cross-compiling tool-chain each one respectively can generate code for running into x86 or into ARM machines.

![Figure 68 Native Toolchain and Cross Compilation Toolchain Definition. Image extracted from [21]](image-url)
The main reason for this tool to exist is that it is not practical that embedded systems spend memory space and CPU for compiling the software that will execute in the future. So the developer has to assume that he is building applications targeting the architecture of his embedded system, in our case ARM.

Figure 69 shows the different shorts of toolchain procedures to build toolchains and binaries for a target machine, being the possibility that the architecture of the target is the same of the host machine\(^3\) and the build machine\(^4\) or not.

![Figure 69 Different Toolchain Build Procedures. Image extracted from [21]](image)

### 4.3.1.1 Components
- **Binutils:** Set of tools to generate and manipulate binaries for a given CPU architecture
  - ld, the GNU linker.
  - as, the GNU assembler.
  - Also include other tools.
- **Kernel headers:** The C library and compiled programs needs to interact with the kernel. Available in include/ folder of the kernel sources
- **C/C++ libraries:** uClibc, lightweight C library for small embedded systems
  - High configurability through a menuconfig interface.
  - Focused on size rather than performance.
  - Actively maintained.
  - Supported and used by MontaVista\(^5\), TimeSys\(^6\) and Wind River\(^7\).

---

\(^3\) Host machine: Machine where the toolchain will be executed.
\(^4\) Build machine: Machine where toolchain is built, could not be the same machine where the toolchain will be executed.
- GCC compiler: The Gnu Compiler Collection, it can compile C, C++, Ada, Fortran, Java, etc. and generate code for a large number of CPU architectures.

As said in 4.3.3, the compilation toolchain that comes with the system runs on and generates code for the processor in the host system. As our embedded system has a different processor, a cross-compilation toolchain is needed. Buildroot provides different toolchains’ solutions.

- Buildroot internal toolchain
- External toolchain
- Crosstool-NG toolchain

When configuring Buildroot, one of these toolchains can be selected using the Toolchain Type option in the Toolchain menu as shown in Figure 75.

4.3.2 Folders at a glance
Buildroot output is stored in a single directory named output/ as depicted in Figure 70. This directory contains several subdirectories:

- images/ where all the images are stored (kernel image, root filesystem).
- build/ where all the components except for the cross-compilation toolchain are built. This includes tools needed to run Buildroot in the host and packages compiled for the target. The build/ directory contains one subdirectory for each of these components.
- staging/ contains a hierarchy similar to a root filesystem hierarchy. This directory contains the installation of the cross-compilation toolchain and all the userspace packages selected for the target. However, it is directory is not intended to be the root filesystem for the target. It contains a lot of development files. This development files are used to compile libraries and applications for the target that depend in other libraries.
- target/ contains almost the complete root filesystem for the target, except the device files in /dev/. As [19] manual says: “Buildroot can’t create it because Buildroot does not run as root and does not want to run as root”. But this directory is not intended to be used on the target; the filesystem for the target is in images/.
- host/ contains the installation of tools compiled for the host that are needed for the proper execution of Buildroot, including the cross-compilation toolchain.
- toolchain/ contains the build directories for the various components of the cross-compilation toolchain.

5 MontaVista: http://www.mvista.com/
6 TimeSys: http://www.timesys.com/
7 Wind River: http://www.windriver.com/
The output/ directory is not the only one that hangs from the top Buildroot folder, the other directories are going to be described above.

- **toolchain/ directory** contains the Makefiles and associated files for all software related to the cross-compilation toolchain, binutils, gcc, kernel-headers and uClibc.
- **arch/** contains the definitions for all the processor architectures that Buildroot supports.
- **package/** contains the Makefiles and associated files for all user-space tools and libraries that Buildroot can compile and add to the target root filesystem.
- **linux/** contains the Makefiles and associated files for Linux kernel.
- **boot/** contains Makefiles and associated files for the bootloaders supported by Buildroot.
- **system/** contains support for the system integration.
- **fs/ directory** contains Makefiles and associated files for software related to the generation of the target root filesystem image.
- **dl/ is the folder where Buildroot locate the files that downloads from the repositories chosen in the configuration.**

### 4.3.3 First configuration

For each menu entry in the configuration tool, you can find associated help that describes the purpose of the entry, that help is materialized in a file named `Kconfig`. The first configuration will be the slowest one because all the target packets have to be downloaded from the specified source, configured, built and installed as schematize Figure 71.
Buildroot has a configuration similar to Linux kernel or Busybox. To run Qt-based configurator

Typing the following command, the Linux shell, a new pop-up will appear. The new window is divided in three parts, on the left part there are the main options pane, some of them with sub-options in the same pane and other with their options on the right part of the window. The lower part is reserved for the information read from `Kconfig` file.

```
$ make xconfig
```

First the Target Architecture, shown in Figure 72 and Figure 73, is selected according to our hardware.
Dealing with embedded systems where memory size is capital, there are several Gnu Cross Compiler optimization levels to be selected, as depicted in Figure 74.
As shown in Figure 74 Target ABI is set to EABI. An Application Binary Interface (ABI) describes the low level interface between a computer program and the operative system. All binaries in a system must be compiled with the same ABI and the kernel must understand this ABI. Embedded Application Binary Interface (EABI) specifies standard conventions for file formats, data types, register usage, stack frame organization and function parameter passing of an embedded software program.

The cross compiler tool-chain selected is the BuildRoot internal tool-chain, Figure 75; it is limited to the usage of the uClibc C library. Linux 3.6.x kernel headers selected according to the Linux source selected, it have backwards compatibility for earlier kernel versions.
The GNU Binutils, shown in Figure 76, are a collection of binary tools to generate and manipulate binaries for a given CPU architecture, the main ones are `ld` the GNU linker, and the `as` the GNU assembler. For more information see reference [22].
Figure 77 shows the GNU Cross Compiler versions supported by this version of BuildRoot, gcc 4.6x is the one selected for ZedBoard.

![Buildroot Toolchain GCC version](image)

In the last part of Toolchain options, Build GDB for the Host will be selected for debugging software applications remotely as shown in Figure 78.

![Buildroot Toolchain last options](image)
Figure 79 shows different options for /dev management. On a Linux system, the /dev directory contains special files, called device files, that 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. Devtmpfs is a virtual filesystem inside the Linux kernel. This filesystem is not persistent across reboots; it is filled dynamically by the kernel with a daemon.

The init program is the first userspace program started by the kernel (it has the PID number 1), and is responsible for starting the userspace services and programs. Buildroot allows to use three different types of init systems. The Busybox init program will read the /etc/inittab file at boot to know what to do.

Buildroot provides a default filesystem skeleton under the directory system/skeleton and the developer can customize it to suit his needs.

The init system, depicted in Figure 79, one of the last things it do is to start a getty program that manages physical or virtual terminals, in our case it handles the connection to the serial terminal for logging in a virtual console with the baud rate selected like Figure 80. As shown in Section 3.6 The Device Tree, ttyPS0 corresponds to the UART0 of the Zynq-7000 PS.

---

8 Device Files: Under UNIX-like Operative Systems, each hardware device is treated as a file. A device file is an interface for a device driver that appears in a filesystem as if was an ordinary file.
Figure 80 Buildroot System Configuration: Root fs skeleton and login prompt.

Shown on Figure 81, Root Filesystem type for the target and as seen in 3.6.4:Linux Bootargs, here it is specified the size and the compression method.

Figure 81 Buildroot Filesystem Image type and options.

Linux kernel has to be enabled, as shown in Figure 82, for Buildroot to download it from the source specified and to build it. Digilent provides Linux sources used for the target across Github
repository. For the first build, the configuration of the kernel will be one of the default ones placed in `arch/arm/configs` folder. For the following configurations where part of the Linux configuration will be changed, this option will be changed to `using a custom config file`, and the path of the `.config` file will be specified. Buildroot will add the suffix _defconfig to the `digilent_zed` as depicted in Figure 82.

![Figure 82 Buildroot Kernel Configuration](image)

Figure 82 Buildroot Kernel Configuration

Figure 83 show the output Linux kernel image format chosen, option named `Kernel binary format`, for the target will be the compressed kernel image `zImage`.

![Figure 83 Buildroot Kernel binary format](image)
Then, close the configurator clicking File→ Quit, that will return us to the Linux Shell, now build the sources typing:

$ make

The `make` command performs the following steps:

- Download selected source files
- Configure, build and install the cross-compiling toolchain using in this case, the internal toolchain
- Build and install selected target packages like Busybox and uClibc
- Build a kernel image with a default configuration
- Create a root filesystem in the selected format

Sometimes Buildroot has some problems about using external repositories to download the packages. The way of detecting that short of problems is to explore the `dl` directory, as explained above, there is where Buildroot leave every package downloaded. It is a common problem that Buildroot does not download the Digilent Linux kernel sources properly, so the developer can download them with `git` and then copy the compressed file in the `dl` directory. Buildroot before going to a foreign repository for a package searches if it is already downloaded and built.

Before the first configuration, the cross compiler toolchain is not downloaded so the path of the cross compiler could not be set in the system for the developer to compile his own programs or other software that are not packaged in Buildroot.

The embedded developer probably has to customize every single part of the system to fit the requirements, for guiding through it the next subsections will show how Linux kernel, Busybox and uClibc are customized for the target.

### 4.3.4 Linux kernel Customization

Based on the default configuration named `digilent_zed_defconfig` included in the Digilent Linux sources, `/linux-sources/arch/arm/configs/`. The following images show how the configuration is customized.

Some of the configurations referred to Ethernet support and PTP driver’s support will be explained in the following chapters.

Run:
Figure 84 shows the general view of the configuration interface for the Linux/arm 3.6.0-digilent-13.01.

![Interactive Curses-based Kernel Configurator](image)

**Figure 84 Interactive Curses-based Kernel Configurator**

*General Setup: in Figure 85, Figure 86 and Figure 87*
Development of Embedded Linux Applications Using ZedBoard

Figure 85 Linux kernel menuconfig 1, General Setup

Figure 86 Linux kernel menuconfig 2, General Setup

Figure 87 Linux kernel menuconfig 3, General Setup
Enable Loadable Module Support: in Figure 88

![Enable loadable module support](image)

Figure 88 Linux kernel menuconfig 4, Enable Loadable Module Support

Enable The Block Layer → IO Schedulers: in Figure 89

![Block Layer IO Schedulers](image)

Figure 89 Linux kernel menuconfig 5, Block Layer IO Schedulers

System Type: in Figure 90 and Figure 91

![System Type](image)

Figure 90 Linux kernel menuconfig 6, System Type

System Type → Xilinx Specific Options: in Figure 92

![Xilinx Specific Options](image)

Figure 91 Linux kernel menuconfig 7, System Type
Figure 92 Linux kernel \textit{menuconfig} 8, System Type Xilinx Specific Options

\textbf{Kernel Features: in Figure 93 and Figure 94}

Figure 93 Linux kernel \textit{menuconfig} 9, Kernel Features

Figure 94 Linux kernel \textit{menuconfig} 10, Kernel Features

\textbf{Boot Options: in Figure 95}
Floating Point Emulation: in Figure 96

Userspace Binary Formats: in Figure 97

Power Management Options: in Figure 98

Networking Support → Networking Options: in Figure 99, Figure 100 and Figure 101
Figure 99 Linux kernel menuconfig 15, Networking Options

Figure 100 Linux kernel menuconfig 16, Networking Options
Device Drivers: in Figure 102, Figure 103 and Figure 104
Device Drivers → Generic Driver Options: in Figure 105

Device Drivers → Connector Unified Userspace<->Kernel Space linker: in Figure 106
Device Drivers → Memory Technology Device (MTD) Support: in Figure 107 and Figure 108

Device Drivers → Device Tree and Open Firmware Support: in Figure 109

Device Drivers → Block Devices: in Figure 110
Device Drivers → SCSI Device Support: in Figure 111

Device Drivers → Network Device Support: in Figure 112
Device Drivers → Network Device Support → PHY Device Support and Infrastructure: in Figure 113

![Figure 113 Linux kernel menuconfig 29, PHY Device and Infrastructure](image)

Device Drivers → Network Device Support → Ethernet Driver Support: in Figure 114 and Figure 115

![Figure 114 Linux kernel menuconfig 30, Ethernet Driver Support](image)

![Figure 115 Linux kernel menuconfig 31, Ethernet Driver Support](image)
Device Drivers → **Input Device Support:** in Figure 116

![Figure 116 Linux kernel menuconfig 32, Input Device Support](image)

Device Drivers → **Character Drivers:** in Figure 117

![Figure 117 Linux kernel menuconfig 33, Character Drivers](image)

Device Drivers → **I2C Support:** in Figure 118

![Figure 118 Linux kernel menuconfig 34, I2C Support](image)
Device Drivers → SPI Support: in Figure 119

![SPI Support](image1)

Figure 119 Linux kernel menuconfig 35, SPI Support

Device Drivers → GPIO Support: in Figure 120

![GPIO Support](image2)

Figure 120 Linux kernel menuconfig 36, GPIO Support

Device Drivers → Watchdog Timer Support: in Figure 121

![Watchdog Timer Support](image3)

Figure 121 Linux kernel menuconfig 37, WDT Support
Device Drivers → USB Support: in Figure 122 and Figure 123

Figure 122 Linux kernel menuconfig 38, USB Support

Device Drivers → MMC Support: in Figure 124

Figure 124 Linux kernel menuconfig 40, MC Support
Device Drivers → RTC Support: in Figure 125

Device Drivers → DMA Engine Support: in Figure 126

Device Drivers → Remoteproc Drivers: in Figure 127

Device Drivers → Pmod Support: in Figure 128
File Systems: in Figure 129 and Figure 130

Figure 129 Linux kernel menuconfig 45, File Systems

Kernel Hacking: in Figure 131

Figure 131 Linux kernel menuconfig 47, Kernel Hacking

Cryptographic API: in Figure 132 and Figure 133
4.3.5 Busybox Customization

After the initial compilation of Buildroot without customizing Busybox, it is time to select the utilities for the host. The following command launches the configuration tool.

```
$ make busybox-menuconfig
```
Figure 135 shows the top menu interactive configurator for Busybox, and the following graphics will illustrate the options selected, compiled and installed in the target to date.

![Busybox Settings Interactive Curses-based Busybox Configurator]

**Busybox Settings→General configuration: in Figure 136**
Busybox Settings $\rightarrow$ Build Options: in Figure 137

Busybox Settings $\rightarrow$ Debugging Options: in Figure 138

Busybox Settings $\rightarrow$ Debugging Options $\rightarrow$ Additional debugging library: in Figure 139
Busybox Settings → Installation Options, `make install` behaviour: in Figure 140

![Figure 140 Busybox menuconfig 5, Installation Options](image1)

Coreutils: in Figure 141, Figure 142, Figure 143, Figure 144, Figure 145 and Figure 146

![Figure 141 Busybox menuconfig 6, Coreutils](image2)

![Figure 142 Busybox menuconfig 7, Coreutils](image3)
Figure 143 Busybox menuconfig 8, Coreutils

Figure 144 Busybox menuconfig 9, Coreutils
Figure 145 Busybox menuconfig 10, Coreutils

Figure 146 Busybox menuconfig 11, Coreutils

Console Utilities: in Figure 147
Debian Utilities: in Figure 148

Editors: in Figure 149
Finding Utilities: in Figure 150 and Figure 151

Figure 150 Busybox menuconfig 15, Finding Utilities

Init Utilities: in Figure 152

Figure 152 Busybox menuconfig 17, Init Utilities

Logging/Password Management Utilities: in Figure 153 and Figure 154
Figure 153 Busybox `menuconfig` 18, Logging/Password Management Utilities

Figure 154 Busybox `menuconfig` 19, Logging/Password Management Utilities

Linux Ext2 FS Progs: in Figure 155

Figure 155 Busybox `menuconfig` 20, Linux Ext2 FS Programs

Linux Module Utilities: in Figure 156
**Figure 156 Busybox menuconfig 21, Module Utilities**

*Linux System Utilities: in Figure 157, Figure 158 and Figure 159*

**Figure 157 Busybox menuconfig 22, System Utilities**
Figure 158 Busybox menuconfig 23, System Utilities

Figure 159 Busybox menuconfig 24, System Utilities

Miscellaneous Utilities: in Figure 160, Figure 161, Figure 162 and Figure 163
Figure 160 Busybox menuconfig 25, Miscellaneous Utilities

Figure 161 Busybox menuconfig 26, Miscellaneous Utilities
Networking Utilities: in Figure 164, Figure 165, Figure 166, Figure 167, Figure 168 and Figure 169
Figure 164 Busybox *menuconfig* 29, Networking Utilities

Figure 165 Busybox *menuconfig* 30, Networking Utilities
BuildRoot, the Build System.

Figure 166 Busybox menuconfig 31, Networking Utilities

Figure 167 Busybox menuconfig 32, Networking Utilities

Figure 168 Busybox menuconfig 33, Networking Utilities
Process Utilities: in Figure 170 and Figure 171

Shells: in Figure 172
4.3.6 uClibc Customization

uClibc is a C library for developing embedded Linux systems. It is smaller than the GNU C, library normally used with Linux distributions, but nearly all applications supported by glibc also work with uClibc.

In the first Buildroot compilation, the default configuration for the library is used, as shown in Figure 76 Buildroot Toolchain: uClibc and Binutils version. Buildroot adds the way of configuring this library across a curses-based configurator with the following command.

```
$ make uclibc-menuconfig
```

Figure 174 shows the top of the configuration menu. The next set of captions describes the configuration used up to date.
Figure 174 Interactive Curses-based uClibc Configurator

Target Architecture Features and Options: in Figure 175

Figure 175 uClibc menuconfig 1, Features and Options

Target Architecture Features and Options → Linux Kernel Header Location: in Figure 176

Figure 176 uClibc menuconfig 2, Linux Kernel Header Location

General Library Settings: in Figure 177, Figure 178 and Figure 179
Figure 177 uClibc menuconfig 3, General Library Settings

Figure 178 uClibc menuconfig 4, General Library Settings

Figure 179 uClibc menuconfig 5, General Library Settings

Advanced Library Settings: in Figure 180
Network Support: in Figure 181

String and Stdio Support: in Figure 182 and Figure 183
Big and Tall: in Figure 184

Library Installation Options: in Figure 185

Security Options: in Figure 186

uClibc Development/debugging options: in Figure 187
4.3.7 Filesystem Customization

Buildroot provides several ways of filesystem customization. As mentioned in 4.3.2 Folders at a glance, the target filesystem is available under output/target/. So if the developer add or delete folders and run make afterwards, this will rebuild the target filesystem image.

Other way is to create a custom target skeleton as depicted in Figure 189. Starting with a default skeleton model available under system/skeleton and then customizing it to fit the needs.

For actual requirements of ZedBoard Linux the Root-filesystem is built like Figure 189.
Chapter 5: 1588 Precision Time Protocol

5.1 Introduction

The Precision Time Protocol (PTP) is a protocol used to synchronize distributed clocks through a computer Local Area Network achieving an accuracy of less than one microsecond. This protocol is suitable for measurement and control distributed systems.

The Gigabit Ethernet MAC controller of the Zynq-7000 SoC has a Timestamp Unit and the recognition of 1588 rev. 2 PTP frames implemented. Having those capabilities and a Linux running in its processors Zynq-7000 in ZedBoard platform is a good candidate a PTP implementation.

5.2 Protocol Description

The Precision Time Protocol provides a medium by which networked computer systems can precisely synchronize to a master clock reference time and estimate their offset from that master clock time through the exchange of special PTP frames over a network. The PTP messages can be transported over Ethernet, over Internet Protocol Version 4 or over Internet Protocol Version 6.

The protocol determines a unique master among a group of clocks using the Best Master Clock Algorithm (BMCA) which selects the most accurate and stable clock of the network.

PTP master send sync messages, the master time stamp the send time of the sync messages and slaves record de receipt time. The difference between the send and the receive times of the sync message is the master to slave delay.

PTP slaves send delay_request messages, slaves record the send time of the delay_request messages and the master records the receipt time. The difference between the send and the receipt times of delay_request messages is the slave to master delay.

PTP calculates an estimate of the propagation delay assuming symmetric propagation delay. And PTP estimates the time difference between master and slave clocks, the master to slave delay corrected for message propagation delay, and it is referred to as the offset from master. Having enough information the clocks will be adjusted as seen below.

For more information refer to [23].

5.3 Zynq-7000 Hardware Support

5.3.1 Gigabit Ethernet Controller

Zynq-7000 Processing System is equipped with two Gigabit Ethernet Controllers; each controller can be configured independently. They can be accessed via MIO and EMIO interfaces. Figure 190 shows
the system viewpoint of the PS Ethernet controllers. For more details review the device datasheet [1] chapter 16.

![Figure 190 Zynq-7000 Gigabit Ethernet Controller System View Point, image extracted from [1]](image_url)

### 5.3.2 IEEE 1588 Time Stamping Unit (TSU)

The TSU consists of a timer and registers to capture the time at which PTP event frames cross the message time-stamping point. The Ethernet MAC provides timestamp registers that capture the departure time and the arrival time of PTP even packets.

The timestamp unit has a 62-bit timer, lower 30 bits for nanoseconds and upper 32 bits for seconds. Every clock cycle, the timer is incremented by a programmable value host in the 1588 timer increment register. The timer is clocked by CPU_1x supposedly derived from the CPU clock. One important drawback is that there does not exist any option for change the clock input or choice freely the clock frequency for this unit.

As can be seen in Figure 190, Time Stamp Unit is connected to the Gigabit Ethernet Controllers and can be routed through the Multiplexed Input Output (MIO) or through the Extended Multiplexed Input Output (EMIO) for the Zynq-7000 Programmable Logic side.
5.3.2.1 IEEE1588 Timestamping Unit Constraints

- There is no hardware access to the counter value and there is not Pulse Per Second output signal for tests.
- The MAC doesn’t provide any capability for implementing programmable alarms when the counter reaches a specific value.
- There is no hardware mechanism provided for synchronizing the counter values of each Ethernet controllers or for making one counter slave to the other.
- There is no support for multiple-queues so there are only two queues in memory, one for received packets and other for transmitted packets.
- The timestamp registers are 1-deep, there is not any FIFO for queuing events.

5.3.3 IEEE 1588 Register Overview

The registers available in the MAC related with the IEEE1588 are summarized in Table 2.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Address</th>
<th>Width</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer_strobe_s</td>
<td>0x000001C8</td>
<td>32</td>
<td>rw</td>
<td>0x00000000</td>
<td>1588 timer sync strobe seconds</td>
</tr>
<tr>
<td>timer_strobe_ns</td>
<td>0x000001CC</td>
<td>32</td>
<td>mixed</td>
<td>0x00000000</td>
<td>1588 timer sync strobe nanoseconds</td>
</tr>
<tr>
<td>timer_s</td>
<td>0x000001D0</td>
<td>32</td>
<td>rw</td>
<td>0x00000000</td>
<td>1588 timer seconds</td>
</tr>
<tr>
<td>timer_ns</td>
<td>0x000001D4</td>
<td>32</td>
<td>mixed</td>
<td>0x00000000</td>
<td>1588 timer nanoseconds</td>
</tr>
<tr>
<td>timer_adjust</td>
<td>0x000001D8</td>
<td>32</td>
<td>mixed</td>
<td>0x00000000</td>
<td>1588 timer adjust</td>
</tr>
<tr>
<td>Timer_incr</td>
<td>0x000001DC</td>
<td>32</td>
<td>Mixed</td>
<td>0x00000000</td>
<td>1588 timer increment</td>
</tr>
<tr>
<td>Ptp_tx_s</td>
<td>0x000001E0</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>PTP event frame transmitted seconds</td>
</tr>
<tr>
<td>Ptp_tx_ns</td>
<td>0x000001E4</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>PTP event frame transmitted nanoseconds</td>
</tr>
<tr>
<td>Ptp_rx_s</td>
<td>0x000001E8</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp event frame received seconds</td>
</tr>
<tr>
<td>Ptp_rx_ns</td>
<td>0x000001EC</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp event frame received nanoseconds</td>
</tr>
<tr>
<td>Ptp_peer_tx_s</td>
<td>0x000001F0</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp peer frame transmitted seconds</td>
</tr>
<tr>
<td>Ptp_peer_tx_ns</td>
<td>0x000001F4</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp peer frame transmitted nanoseconds</td>
</tr>
<tr>
<td>Ptp_peer_rx_s</td>
<td>0x000001F8</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp peer frame received seconds</td>
</tr>
<tr>
<td>Ptp_peer_rx_ns</td>
<td>0x000001FC</td>
<td>32</td>
<td>ro</td>
<td>0x00000000</td>
<td>Ptp peer frame received nanoseconds</td>
</tr>
</tbody>
</table>
5.4 Linux Drivers Support

Linux sources incorporate partial support for IEEE1588 PTP clocks to standardize the method for developing PTP user space programs, the infrastructure registers with a class driver that creates a character device for each registered clock and supports a complete set of PTP hardware clock functionality as described in Linux Sources folder `/Documentation/ptp` [12] of the kernel source tree. Up to date, the hardware clocks supported by Linux are:

- Freescale eTSEC gianfar.
- National Semiconductor DP83640
- Intel IXP465

One of the objectives of this work has been to add PTP support for Zynq-7000 in Linux. The following sections will guide through the process followed and expound what is needed and what is already done.

5.4.1 Adding Linux Kernel Support for PTP

For adding support for PTP on the Zynq-7000 Linux embedded distribution created, BuildRoot Linux Kernel configurator have to be launched as shown in Figure 191.

```
$ make linux-menuconfig
```

![Figure 191 Kernel PTP 1588 Clock Support](image)
PTP options in the kernel configurator are placed in **Device Drivers→ PTP clock support**, as shown in Figure 191. Entering in **PTP clock support**, the only option allowed to select is the one shown in Figure 192.

![Figure 192 Enable Linux Kernel PTP 1588 Clock Support](image)

If the **Help** of the **PTP clock support** is displayed, as depicted in Figure 193, a description guides the selection of the options. “If you want to use a PTP clock, then you should also enable at least one of the clock driver as well”, for Zynq-7000 there is no Clock Driver support, therefore that part have to be codified and built. **Select PTP clock support** option and **exit** menuconfig.
5.4.2 PTP in the Linux Sources

To understand what have been added in the previous section, some files of the Linux Sources are going to be explored. As seen in Chapter 4: BuildRoot, the Build System. The Linux Kernel sources are placed in buildroot_2012_11_1/output/build/linux-digilent.

If you navigate to /drivers/ptp and open the Makefile for PTP 1588 clock support as shown in Listing 5, there can be seen that the objects that are going to be added in the Kernel are ptp_clock.o, ptp_chardev.o and ptp_sysfs.o, objects that come from ptp_clock.c, ptp_chardev.c and ptp_sysfs.c respectively, in an object named ptp.o.

Listing 5 Makefile PTP Clock Support

```makefile
# Makefile PTP Clock Support.
```
That `ptp.o` implements a PTP Device Driver to handle the PTP clock or clocks. As explain the Linux PTP documentation placed in `Buildroot_2012_11_1/output/build/linux-digilent/Documentation/ptp/``A PTP clock driver registers itself with the class driver ... The class driver handles all of the dealings with user space...” [12].

What it is intended, it is to create a two-layer structure driver as depicted in Figure 194. The class driver, being the top layer, does not take the responsibility for interacting with the hardware. The duties of the class driver are:

- Create a character device for each registered clock in `/dev/` folder of the Filesystem
- Register the driver in `sysfs`.
- Provide the capabilities of the hardware for the User Space to interact with it.
- Provide an abstraction layer between the bottom layer, who manages the hardware, and the user space.
Linux provides a procedure for managing PTP Hardware Clocks (PHCs), that procedure can be shown in Buildroot_2012_11_1/output/build/linux-digilent/include/linux/ptp_clock_kernel.h header file shown in Listing 6.

Listing 6 ptp_clock_kernel.h (Copyright © 2010 OMICRON electronics GmbH)

```c
/*
 * PTP 1588 clock support
 */
...

#define _PTP_CLOCK_KERNEL_H_

#include <linux/ptp_clock.h>

struct ptp_clock_request {
    enum {
        PTP_CLK_REQ_EXTTS,
        PTP_CLK_REQ_PEROUT,
        ...
    }
```
```c
PTP_CLK_REQ_PPS,
}
union {
    struct ptp_extts_request extts;
    struct ptp_perout_request perout;
};

/**
 * struct ptp_clock_info - describes a PTP hardware clock
 * @owner:     The clock driver should set to THIS_MODULE.
 * @name:      A short name to identify the clock.
 * @max_adj:   The maximum possible frequency adjustment, in parts per billion.
 * @n_alarm:   The number of programmable alarms.
 * @n_ext_ts:  The number of external time stamp channels.
 * @n_per_out: The number of programmable periodic signals.
 * @pps:       Indicates whether the clock supports a PPS callback.
 *
 * clock operations
 *
 * @adjfreq:  Adjusts the frequency of the hardware clock.
 *           parameter delta: Desired period change in parts per billion.
 *
 * @adjtime:  Shifts the time of the hardware clock.
 *           parameter delta: Desired change in nanoseconds.
 *
 * @gettime:  Reads the current time from the hardware clock.
 *           parameter ts: Holds the result.
 *
 * @settime:  Set the current time on the hardware clock.
 *           parameter ts: Time value to set.
 *
 * @enable:   Request driver to enable or disable an ancillary feature.
 *           parameter request: Desired resource to enable or disable.
 *           parameter on: Caller passes one to enable or zero to disable.
 *
 * Drivers should embed their ptp_clock_info within a private structure, obtaining a reference to it using container_of().
 *
 * The callbacks must all return zero on success, non-zero otherwise.
 */

struct ptp_clock_info {
    struct module *owner;
    char name[16];
    s32 max_adj;
    int n_alarm;
    int n_ext_ts;
    int n_per_out;
    int pps;

    int (*adjfreq)(struct ptp_clock_info *ptp, s32 delta);
    int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
    int (*gettime)(struct ptp_clock_info *ptp, struct timespec *ts);
};
```
int (*settime)(struct ptp_clock_info *ptp, const struct timespec *ts);
int (*enable)(struct ptp_clock_info *ptp,
               struct ptp_clock_request *request, int on);

struct ptp_clock;

/**
 * ptp_clock_register() - register a PTP hardware clock driver
 * @info: Structure describing the new clock.
 */
extern struct ptp_clock *ptp_clock_register(struct ptp_clock_info *info);

/**
 * ptp_clock_unregister() - unregister a PTP hardware clock driver
 * @ptp: The clock to remove from service.
 */
extern int ptp_clock_unregister(struct ptp_clock *ptp);

As can be seen in Listing 6 struct ptp_clock_info describes the capabilities of the PTP Hardware Clock: adjfreq, adjtime, settime, gettime and enable. The structure of the PTP Hardware Clock is defined in Buildroot_2012_11_1/output/build/linux-digilent/drivers/ptp/ptp_private.h as shown in Listing 7.

Listing 7 Definition of PTP Hardware Clock Structure, Extract of ptp_private.h (Copyright © 2010 OMICRON electronics GmbH)
As can be seen in Line 44 of Listing 7 the PTP Hardware Clock structure `struct ptp_clock` incorporates a pointer to the structure `ptp_clock_info` which contains the data structure that the PTP Hardware Clock driver has to manage.

5.4.3 Communication paradigm between the two driver layers

Commanding a clock operation of the driver like `adjtime`, `adjfreq`, `gettime`, `settime` or `enable` methods, will cause the execution of the part of the PTP class driver code which accomplishes the operation requested. That execution of the fragment of the class driver will cause the execution of the fragment of code from the PTP Hardware Clock driver that performs the `adjtime`, `adjfreq`, `gettime`, `settime` or `enable`. If some data has to be returned back to the commander, it will be sent from the PTP Hardware Clock driver to the class driver and from the class driver to the commander of the operation.

Listing 8 shows an extract of the `ptp_clock.c` file, one of the parts of the class driver which could be found in [12]/drivers/ptp/.

To access to the `settime` method of the PTP Hardware Clock driver and finally write to the adjust value in physical register of the device, the kernel helper function `container_of()` is used to elicit the address of the container structure. On Listing 8 Line 112 it is shown the access to the `settime` function implemented in the PTP Hardware Clock driver as seen in Listing 9.

Listing 8 Extract of ptp_clock.c Class Driver

```c
109 static int ptp_clock_settime(struct posix_clock *pc, const struct timespec *tp)
110 {
111     struct ptp_clock *ptp = container_of(pc, struct ptp_clock, clock);
112     return ptp->info->settime(ptp->info, tp);
113 }
```

```
155 static struct posix_clock_operations ptp_clock_ops = {
156     .owner = THIS_MODULE,
157     .clock_adjtime = ptp_clock_adjtime,
158     .clock_gettime = ptp_clock_gettime,
159 };
```
Listing 9 Extract of PTP Hardware Clock Driver Implementation

...  
1362 /* @settime: Set the current time on the hardware clock.  
1363   *            parameter ts: Time value to set.  
1364 */  
1365 static int ptp_xemacps_settime(struct ptp_clock_info *ptp, const struct timespec *ts)  
1366 {  
1367   u64 ns;  
1368   unsigned long flags;  
1369   struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct tyxemacps_clock, caps);  
1370   ns = ts->tv_sec * 1000000000ULL;  
1371   ns += ts->tv_nsec;  
1372   spin_lock_irqsave(&register_lock, flags);  
1373   xemacps_write_ptp_clock(xemacps_clock, ns);  
1374   spin_unlock_irqrestore(&register_lock, flags);  
1375   return 0;  
1376 }  
...  
...  
1541 static struct ptp_clock_info ptp_xemacps_caps = {  
1542   .owner = THIS_MODULE,  
1543   .name = XEMACPS Hardware Clock",  
1544   .max_adj = 66666655,  
1545   .n_ext_ts = N_EXT_TS,  
1546   .pps = 0,  
1547   .adjfreq = ptp_xemacps_adjfreq,  
1548   .adjtime = ptp_xemacps_adjtime,  
1549   .gettime = ptp_xemacps_gettime,  
1550   .settime = ptp_xemacps_settime,  
1551   .enable = ptp_xemacps_enable,  
1552 };  
...
In this example it is shown the interconnection mechanism between the two layers of the driver. To illustrate the mechanism Figure 195 justifies the use of `container_of()` function. The class driver uses the `posix_clock` data structures, so to access to the PTP Hardware Clock (PHC) the reference of the `ptp_clock` structure has to be obtained and then `ptp_clock_info` structure is accessible.

![Figure 195 Device Driver Data Structures Diagram](image)

Once that the main design aspects of the PHC driver had been explained the next section will delve in the technical details and construction.

### 5.5 Xilinx Ethernet Driver, The `xilinx_emacps`

#### 5.5.1 Linux Support for IEEE1588 in the Xilinx Ethernet Driver

To enable the driver functionality of Generate Hardware Packet Timestamps for the Ethernet driver of the Zynq-7000 Linux embedded distribution created, BuildRoot Linux Kernel configurator have to be launched.

```
$ make linux-menuconfig
```

Ethernet driver selection options in the kernel configurator are placed in Device Drivers→ Network device support→ Ethernet driver support, as shown in Figure 196. The Ethernet driver selected for Zynq-7000 is Xilinx PS tri-speed EMAC which corresponds to `xilinx_emacps.c` file built as
seen in Listing 10. The Ethernet driver will be composed by three objects `ll_tmac_main`, `temac_mdio` and `xilinx_emacps`.

As depicted in Figure 196, selecting Xilinx PS tri-speed EMAC allow to select a sub-option for hardware packet timestamping. Figure 197 shows the Kconfig help about that sub-option.
Exploring the code, placed in Buildroot_2012_11_1/output/build/linux-digilent/drivers/net/Ethernet/xilinx, enabling XILINX_PS_EMAC_HWTSTAMP will set to true-logic a #define constant called CONFIG_XILINX_PS_EMAC_HWTSTAMP which permit some parts of the driver been built.

Select PTP clock support option, exit menuconfig. Back in the Linux Shell typing make will perform a rebuild of the Linux Kernel sources to incorporate the changes.

5.5.2 PHC in the Xilinx Ethernet Driver

Analysing the methods implemented within the xilinx_emacps file enabled with XILINX_PS_EMAC_HWTSTAMP, the decision was to create the PTP Hardware Clock in the same file for testing it and then export the created code to make a patch applicable to the xilinx_emacps Ethernet driver.

The code inserted is merged under XILINXPTP constant.

The first thing added is the header linux/ptp_clock_kernel.h which hosts the data types for supporting the PTP Hardware Clock.
Listing 11 ptp_clock_info Structure Declared in ptp_clock_kernel.h

```c
struct ptp_clock_info {
    struct module *owner;
    char name[16];
    s32 max_adj;
    int n_alarm;
    int n_ext_ts;
    int n_per_out;
    int pps;
    int (*adjfreq)(struct ptp_clock_info *ptp, s32 delta);
    int (*adjtime)(struct ptp_clock_info *ptp, s64 delta);
    int (*gettime)(struct ptp_clock_info *ptp, struct timespec *ts);
    int (*settime)(struct ptp_clock_info *ptp, const struct timespec *ts);
    int (*enable)(struct ptp_clock_info *ptp, struct ptp_clock_request *request, int on);
};
```

A new data type has been defined, tyxemacps_clock, as seen in Listing 12, which merges the ptp_clock structure and the ptp_clock_info structure and also a control field. Those methods that are inside the ptp_clock_info structure will be defined later.

Listing 12 Clock Structure for Management, Host in xilinx_emacps

```c
typedef struct tyxemacps_clock {
    struct ptp_clock *ptp_clock;
    struct ptp_clock_info caps;
    void __iomem *ethbaseaddress;
    int exts0_enabled;
};

struct tyxemacps_clock xemacps_clock;
```

Listing 13 shows how the structure ptp_clock_info is set up.

Listing 13 Set up of the PHC Structure

```c
static struct ptp_clock_info ptp_xemacps_caps = {
    .owner = THIS_MODULE,
    .name = "XEMACPS timer",
    .max_adj = 66666655,
    .n_ext_ts = N_EXT_TS,
    .pps = 0,
    .adjfreq = ptp_xemacps_adjfreq,
    .adjtime = ptp_xemacps_adjtime,
    .gettime = ptp_xemacps_gettime,
};
```
The registration of the clock is made in xemacps_probe, the probe function of the Ethernet driver. As seen in Listing 14, the driver calls the function ptp_clock_register() provided by the ptp class driver (ptp_clock.c under ../drivers/ptp/), which sets up all the data structures and creates the char driver under /dev/.

Listing 14 Registration of the PHC

```c
xemacps_clock.ptp_clock = ptp_clock_register(&xemacps_clock.caps);
```

The ioremap() function returns a virtual address that can be used to access the specified physical address range. The pointer returned is kept in one of the fields of net_local, the private Ethernet device data structure and additionally in the tx_emacps_clock PHC structure for accessing to the Zynq-7000 PTP registers.

Listing 15 Obtaining the Virtual Address of the Ethernet Controller

```c
lp->baseaddr = ioremap(r_mem->start, (r_mem->end - r_mem->start + 1));
...

xemacps_clock.ethbaseaddress = lp->baseaddr;
```

Some of the methods already implemented by Xilinx in xilinx_emacps for managing the PTP registers like xemacps_read_ptp_clock, xemacps_write_ptp_clock, xemacps_set_ptp_hwticks, xemacps_get_ptp_hwticks are used for gettime, settime, adjtime, enable and adjfreq.

As can be seen in Listing 16, xemacps_read_ptp_clock and xemacps_get_ptp_hwticks use as input structure the tx_emacps_clock, the reason is that for writing and reading from the hardware registers the virtual address of the Ethernet device, one of the fields of tx_emacps_clock as seen in Listing 12, is needed. XEMACPS_1588S_OFFSET and XEMACPS_1588NS_OFFSET are the register offsets for the IEEE1588 Timer Seconds and IEEE1588 Timer Nanoseconds registers.

Listing 16 xemacps_read_ptp_clock and xemacps_get_hwticks Functions for Supporting PTP Hardware Control
static u64 xemacps_read_ptp_clock(struct txmacps_clock *xemacps_clock)
{
    u64 stamp;
    u64 sec, nsec;
    xemacps_get_ptp_hwticks(xemacps_clock, &sec, &nsec);
    stamp = (sec << 32) | nsec;
    return stamp;
}

static inline void xemacps_get_ptp_hwticks(struct txmacps_clock *xemacps_clock, u64 *sec, u64 *nsec)
{
    do {
        *nsec = xemacps_read(xemacps_clock->ethbaseaddress, XEMACPS_1588NS_OFFSET);
        *sec = xemacps_read(xemacps_clock->ethbaseaddress, XEMACPS_1588S_OFFSET);
    } while (*nsec > xemacps_read(xemacps_clock->ethbaseaddress, XEMACPS_1588NS_OFFSET));
}

Listing 17 shows the functions for writing in the IEEE1588 Timer Seconds and IEEE1588 Timer Nanoseconds registers. As explained for Listing 16, one of the fields of the txmacps structure is the Ethernet virtual direction needed for accessing to the hardware.

Listing 17 xemacps_write_ptp_clock and xemacps_set_ptp_hwticks Functions for Supporting PTP Hardware Control

static u64 xemacps_write_ptp_clock(struct txmacps_clock *xemacps_clock, u64 ns)
{
    u64 sec;
    u64 nsec;
    sec = ns >> 32;
    nsec = ns & 0xffffffff;
    xemacps_set_ptp_hwticks(xemacps_clock, &sec, &nsec);
    return 0;
}

static int xemacps_set_ptp_hwticks(struct txmacps_clock *clock, u64 *sec, u64 *nsec)
{
    xemacps_write(clock->ethbaseaddress, XEMACPS_1588NS_OFFSET, *nsec);
    xemacps_write(clock->ethbaseaddress, XEMACPS_1588S_OFFSET, *sec);
    return 0;
The following subsections will describe the methods for managing the PHC clock, `gettime`, `settime`, `enable`, `adjtime` and `adjfreq`. For the rest of the implementation the appendix will contain the source file.

### 5.5.2.1 Gettime

As described in the IEEE1588 Standard and as can be seen in [1] the PTP clock has to be composed by a 32-bit Timer for seconds and a 30-bit Timer for nanoseconds. The method shown in Listing 18 implements a read of the 1588 Timer Seconds and 1588 Timer Nanoseconds, value read is written to the `timespec` structure. As explained in `ptp_clock_kernel.h` the access to the structure is made with the function `container_of()`.

#### Listing 18 Driver Method `gettime`

```c
/*
 * @gettime: Reads the current time from the hardware clock.
 * parameter ts: Holds the result.
 */
static int ptp_xemacps_gettime(struct ptp_clock_info *ptp, struct timespec *ts)
{
    u64 ns;
    u32 remainder;
    unsigned long flags;
    struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct tyxemacps_clock, caps);
    spin_lock_irqsave(&register_lock, flags);
    ns = xemacps_read_ptp_clock(xemacps_clock);
    spin_unlock_irqrestore(&register_lock, flags);
    ts->tv_sec = div_u64_rem(ns, 1000000000, &remainder);
    ts->tv_nsec = remainder;
    return 0;
}
```

### 5.5.2.2 Settime

Like `gettime`, `settime` method, described in Listing 19, mission is to access to the seconds and nanoseconds 62 Timer for writing the value specified by the `timespec` structure.
Listing 19 Driver Method *settime*

```c
/*
 * @settime:  Set the current time on the hardware clock.
 *            parameter ts: Time value to set.
 */
1 static int ptp_xemacps_settime(struct ptp_clock_info *ptp, const struct timespec *ts)
2 {
3   u64 ns;
4   unsigned long flags;
5   struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct
tyxemacps_clock, caps);
6   7   ns = ts->tv_sec* 1000000000ULL;
8   ns += ts->tv_nsec;
9   10  spin_lock_irqsave(&register_lock, flags);
11  xemacps_write_ptp_clock(xemacps_clock, ns);
12  spin_unlock_irqrestore(&register_lock, flags);
13  14  return 0;
15 }
```

5.5.2.3 Adjtime

This method, described in Listing 20, implements a read access and a write access to the IEEE1588 62-bit Timer registers of the Zynq-7000. The seconds and nanoseconds read value is incremented or decremented by the input parameter delta and written back again to the registers.

Listing 20 Driver Method adjtime

```c
/*
 * @adjtime:  Shifts the time of the hardware clock.
 *            parameter delta: Desired change in nanoseconds.
 */
1 static int ptp_xemacps_adjtime(struct ptp_clock_info *ptp, s64 delta)
2 {
3   s64 now=0x0;
4   unsigned long flags;
5   struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct
tyxemacps_clock, caps);
6   7   spin_lock_irqsave(&register_lock, flags);
8   now = xemacps_read_ptp_clock(xemacps_clock);
9   now += delta;
10  xemacps_write_ptp_clock(xemacps_clock, now);
11  spin_unlock_irqrestore(&register_lock, flags);
12  13  return 0;
14  15 }
```
5.5.2.4 Enable

As explained in Listing 21 this method implements the enable or disable of a feature of the PHC, the hardware implementation of the IEEE1588 protocol on Zynq-7000 is so limited that the only feature that can be enable or disable is the PHC in itself.

Listing 21 Driver Method enable

```c
/*
 * @enable: Request driver to enable or disable an ancillary feature.
 *          parameter request: Desired resource to enable or disable.
 *          parameter on: Caller passes one to enable or zero to disable.
 */
static int ptp_xemacps_enable(struct ptp_clock_info *ptp, struct ptp_clock_request *rq, int on)
{
    struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct tyxemacps_clock, caps);

    switch (rq->type)
    {
        case PTP_CLK_REQ_EXTTS:
            switch (rq->extts.index)
            {
                case 0:
                    xemacps_clock->exts0_enabled = on ? 1 : 0;
                    break;
                default:
                    return -EINVAL;
            }
            return 0;
        default:
            break;
    }
    return -EOPNOTSUPP;
}
```

5.5.2.5 Adjfreq

As described in ptp_clock_kernel.h, the adjfreq method has to adjust the frequency of the PTP Hardware Clock with the value delta expressed in parts per thousand millions.

As shown in Figure 198 the Gigabit Ethernet Controller (GEM) register of Zynq-7000 for accomplishing this adjust is timer_incr.
Register (GEM) timer_incr

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>31:24</td>
<td>ro</td>
<td>0x0</td>
<td>Reserved, read as 0, ignored on write.</td>
</tr>
<tr>
<td>incr_b4_alt</td>
<td>23:16</td>
<td>rw</td>
<td>0x0</td>
<td>The number of increments after which the alternative increment is used.</td>
</tr>
<tr>
<td>alt_ct_ns_delta</td>
<td>15:8</td>
<td>rw</td>
<td>0x0</td>
<td>Alternative count of nanoseconds by which the 1588 timer nanoseconds register will be incremented each clock cycle.</td>
</tr>
<tr>
<td>ns_delta</td>
<td>7:0</td>
<td>rw</td>
<td>0x0</td>
<td>A count of nanoseconds by which the 1588 timer nanoseconds register will be incremented each clock cycle.</td>
</tr>
</tbody>
</table>

Figure 198 1588 Timer Increment Register Details

The register is fractioned in three parts; the following equation induces the behaviour of the timer_increment register.

\[(x + 1) \times T_{clk} = x \times A + B\]

Where \(A\) corresponds to ns_delta, \(B\) corresponds to alt_ct_ns_delta and \(x\) corresponds to incr_b4_alt. For clarifying Figure 199 is an induced explanatory diagram of operation of the timer increment register.
The \textit{ns\_delta} and \textit{alt\_ct\_ns\_delta} 8-bit registers are used as incrementing values for the 1588 timer seconds and 1588 timer nanoseconds registers. Each clock cycle, the 1588 timer seconds and nanoseconds are incremented with the value contained in \textit{ns\_delta} or \textit{alt\_ct\_ns\_delta} registers. The selection of which is used is governed by \textit{inc\_b4\_alt} register, each time the number of increments done with the \textit{ns\_delta} value reach the value contained in \textit{inc\_b4\_alt}, in the next clock cycle the increment will be the value contained in \textit{alt\_ct\_ns\_delta}.

Now the problem begins when the values that contain these registers have to change to adjust the frequency of the 1588 Timer seconds and nanoseconds. Returning to the equation described above \( T_{clk} \) is the unique fixed value. Then the problem is how to solve a three variable polynomial. There are some conditions that the variables have to satisfy, enumerated below.

- As explained above, the expression has to satisfy

\[
\frac{x \cdot A + B}{x + 1} = T_{clk}
\]

- A, B and x values have to be between 1 and \( 2^8 \)
- A, B and x values have to be positive.

The \( T_{clk} \) maximum and minimum values are governed with the following equation; the maximumAdjust value is in parts per 1000 million. The \( T_{clk\_CenterValue} \) is the 1588 timer increment default value for the 1588 timer seconds and nanoseconds to act as a 1 Hz clock.
Now that the possible \( T_{clk} \) range of values are known the goal is to find the combination values of \( \text{ns}_{\text{delta}} \), \( \text{alt}_{\text{ct}} \text{ns}_{\text{delta}} \) and \( \text{inc}_b4_{\text{alt}} \) registers for satisfying the equation.

\[
T_{clk} = \frac{x\cdot A + B}{x + 1}
\]

In mathematics, a Diophantine equation is a polynomial equation with two or more variables that only integer solutions are allowed [24]. Analysing the equation that describes the behaviour of 1588 timer register adjust can be seen that it does not look like a linear Diophantine Equation (first-order Diophantine Equation), it is already a second order Diophantine Equation. The computation time to solve a second order Diophantine equation is extremely high and the process complexity to implement it in a Linux device driver is unreasonable.

An empiric approximation of the equation is been made to facilitate the calculus of each variable. If the \( A \) term that corresponds to \( \text{ns}_{\text{delta}} \) is set to value 2 the Diophantine equation will convert to linear.

\[
\frac{x^2 + B}{x + 1} = T_{clk} \\
\text{yields} \\
B = x \cdot (T_{clk} - 2) + T_{clk}
\]

Seen that the \( T_{clk} \) main value is approximately 2.08 nanoseconds and calculating the maximum upper and lower adjust value.

\[
2.08 - \frac{2.08 \times 32767999}{10^9} = 2.011842562 ...
\]

\[
2.08 + \frac{2.08 \times 32767999}{10^9} = 2.148157438 ...
\]
The resolution of the registers is 8 bits so as depicted in Figure 200, for first approximation of the calculus the lower value for \( \text{inc}_b4\_\text{alt} \) is 3124 and for \( \text{alt}_c\_\text{ct}_n\_\text{ns}_\delta \) is 39 for the maximum lower adjust of the 1588 Timer increment register. So the maximum lower adjust has to be truncated for the solutions to fit in 8-bit registers.

\[
2.01184 = \frac{x \times 2 + y}{x + 1}, \ x > 1, \ y > 1, \ x > y \text{ integer solution}
\]

\[
\text{solving:} \quad x > 1, \ y > 1, \ x > y
\]

\[
\text{Result:} \quad n \neq 0 \text{ and } x = 3125 n + 3124 \text{ and } y = 37 n + 39 \text{ and } n \in \mathbb{Z}
\]

\[
\text{Example of an integer solution:} \quad x = 6249 \text{ and } y = 75
\]

*Figure 200 WolframAlpha Diophantine Equation Resolve for the Maximum Lower Value*

As shown in Figure 201, the maximum lower adjust of the 1588 Timer increment register has to be truncated to 2.01, for that value the solutions that fits the Diophantine linear equation will be the following: the \( \text{inc}_b4\_\text{alt} \) has to be set to 99, \( \text{alt}_c\_\text{ct}_n\_\text{ns}_\delta \) to 3 and \( \text{ns}_\delta \) to 2.
Figure 201 WolframAlpha Diophantine Equation Resolve for the Maximum Lower Value Truncated

For the maximum upper value of adjust happens the same, the value of adjust has to be truncated because as shown in Figure 202 the value 19999 will not fit in an 8-bit register.
As depicted in Figure 203 the maximum upper value is truncated to 2.148, for that value the solutions that fits the Diophantine linear equation will be the following: the inc_b4_alt has to be set to 249, alt_ct_ns_delta to 39 and ns_delta to 2.
Listing 22 show the implementation of the `adjfreq` method for adjusting the frequency of the 1588 timer seconds and nanoseconds register by the implementation of the Diophantine equation.

Listing 22 Driver Method `adjfreq`

```c
/*
 * @adjfreq: Adjusts the frequency of the hardware clock.
 *          parameter delta: Desired period change in parts per billion.
 */
static int ptp_xemacps_adjfreq(struct ptp_clock_info *ptp, s32 delta) {
    u64 diff;
    u32 factor = 1;
    int power = 4; // (greater or equal to 2 for not loosing decimals of nom_adj)
    u32 timerincr_reg = 0x00;
    int def_timerinc_val = 0;
    u32 nom_adj; // nominal adjust
    u32 new_adj; // new value adjust
    int exit = 0;
    s32 a, b, x, b_term, x_term, indep_term; // for diophantine resolve
    int i, neg_adj = 0;
    struct tyxemacps_clock *xemacps_clock = container_of(ptp, struct
```
tyxemacps_clock, caps);

if (delta < 0)
    {
        neg_adj = 1;
        delta = -delta;
    }

a = DEFAULT_1588TIMERINC & 0x0F;
b = (DEFAULT_1588TIMERINC >> 8) & 0xFF;
x = (DEFAULT_1588TIMERINC >> 16) & 0xFF;

if (delta == 32767999) //max_adj
    {
        if (neg_adj==1)
            xemacps_write(xemacps_clock-&ethbaseaddress,
XEMACPS_1588INC_OFFSET,0x00630302); //max lower adj 2.01
        else
            xemacps_write(xemacps_clock-&ethbaseaddress,
XEMACPS_1588INC_OFFSET,0x00F92702); //max upper adj 2.148
    }
else //solve diphantine eq.
    {
        for (i=0;i<power;i++)
            { factor*=10; }

        do{
            nom_adj = (((x*a)+b)*factor)/(x+1);
            diff=(u64)delta*(u64)nom_adj;
            diff = div_u64(diff, 1000000000ULL);

            if(neg_adj)
                new_adj = nom_adj - diff;
            else
                new_adj = nom_adj + diff;

            indep_term = new_adj;
x_term = new_adj - (2 * factor);
b_term = factor;

            while(!(x_term%2) && (x_term%5))
                {
                    if((indep_term%2 == 0) &&(x_term%2 == 0) &&(b_term%2 == 0))
                        { indep_term>>=1;
x_term>>=1;
b_term>>=1; }
                    if((indep_term%5 == 0) &&(x_term%5 ==0) &&(b_term%5 ==0))
                        { indep_term//=5;
x_term//=5; }
5.5.3 Testing the Driver

For testing the functionalities of the implemented PTP driver, Linux sources provide a very modest program testptp. The source file of the program is placed in `<Linux_Sources_folder>/Documentation/ptp`, also it can be found in the Linux Cross Reference [12].

The program has to be built against ARM’s Zynq-7000 architecture and copied inside the Linux Filesystem to execute it.

To cross-compile the testptp program some modifications of the Makefile have to be made. Change the include/linux Linux kernel headers path to the kernel headers path of the target platform, placed in `<buildroot_sources>/output/build/<linux_sources>/include/linux` path. As described in Error! Reference source not found., line 2 the first INC is followed by the # symbol which in Makefile language is to comment code lines, so is replaced for line 3.

```c
b_term/=1;
}

if(((x_term*b_term+indep_term)/b_term)>255 || (b_term>255))
{
    exit = 0;
    if(factor < 100)
    {
        exit = 1;
        def_timerinc_val = 1;
    }
    else
        factor/=10;
    }
    exit=1;
}
while(!exit);

if (def_timerinc_val)
    xemacps_write(xemacps_clock->ethbaseaddress,
    XEMACPS_1588INC_OFFSET,0x00180402); //2.08
else
{
    x = b_term;
    b = (x_term*b_term+indep_term)/b_term;
    a = 2;
    timerincr_reg = ((x<<16) | (b<<8) | a);
    xemacps_write(xemacps_clock->ethbaseaddress,
    XEMACPS_1588INC_Offset,timerincr_reg);
}
return 0;
```
The PATH Linux environment variable has to be set to the path where BuildRoot has host all the binaries for cross-compiling in this case, `<buildroot_sources>/output/host/usr/bin`. In the Linux shell export the environment variable as follows.

```bash
$ export PATH= "<buildroot_sources>/output/host/usr/bin":PATH
```

Then the program can be built running the following command in the path that hosts the sources, `<Linux_Sources_folder>/Documentation/ptp`

```bash
$ make CROSS_COMPILE=arm-linux- ARCH=arm
```

The last step is to export the binary file generated to ZedBoard Linux Filesystem and run it.
5.6 User Space Program

The Linux PTP Project [25] is an implementation of the PTP IEEE1588 for Linux. It supports software and hardware time-stamping.

5.6.1 Getting Sources

The source code can be downloaded using git or in sourceforge platform.

$ git clone git://git.code.sf.net/p/linuxptp/code linuxptp

5.6.2 Compiling Sources

Like the Makefile of testptp, some modifications have to be done for cross-compiling the sources of the linuxptp program.

Listing 24 linuxptp Makefile

```
# Copyright (C) 2011 Richard Cochran <richardcochran@gmail.com>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
1  KBUILD_OUTPUT ?= /lib/modules/$(shell uname -r)/build
2
3  FEAT_CFLAGS :=
4  ifndef $(shell grep clock_adjtime /usr/include/bits/time.h),)
5  FEAT_CFLAGS += -D_GNU_SOURCE -DHAVE_CLOCK_ADJTIME
6  endif
7
8  DEBUG =
9  CC = $(CROSS_COMPILE)gcc
10 #INC = -I$(KBUILD_OUTPUT)/usr/include
11 INC = -
12 VER = -DVER=$(version)
13 CFLAGS = -Wall $(VER) $(INC) $(DEBUG) $(FEAT_CFLAGS) $(EXTRA_CFLAGS)
14 LDLIBS = -lm -lrt $(EXTRA_LDFLAGS)
15 PRG = ptp4l pmc phc2sys hwstamp_ctl
16 OBJ = bmc.o clock.o clockadj.o config.o fault.o fsm.o ptp4l.o mave.o \
17   msg.o phc.o pi.o port.o print.o raw.o servo.o sk.o stats.o tlv.o tmtab.o \
18   transport.o udp.o udp6.o uds.o util.o version.o
19```
Then the program can be built running the following command in the path that hosts the sources,
<linuxptp_Sources_folder>/
As shown in Listing 24, compiling linuxptp will generate four programs, `pmc`, `phc2sys`, `hwstamp_ctl` and `ptp4l`.

**5.6.2.1 Phc2sys**

Phc2sys is a program which synchronizes two clocks in the system. Typically, it is used to synchronize the system clock to a PTP hardware clock (PHC), which itself is synchronized by the `ptp4l` program.

Two synchronization modes are supported, one uses the pulse per second PPS signal provided by the source clock and the other mode reads the time from the source clock directly. The PPS mode is usually preferred, because reading the PHC is slow and introduces an unknown error in the readings, but not all PHCs provide the PPS signal. That is the case of the PHC of the ZedBoard.

**5.6.2.2 Hwstamp_ctl**

Hwstamp_ctl is a program used to set the hardware time stamping policy at the network driver level with the `SIOCSHWTSTAMP` ioctl, for more information refer to [12] timestamping.txt.

The tx-type and the rx-filter values are hints to the driver what is expected to do. If the requested filtering for incoming packets is not supported, the driver may timestamp more than just the requested types of packets.

This program is a debugging tool. The `ptp4l` program does not need this program to function; it will set the policy automatically as appropriate.

**5.6.2.3 Pmc**

PTP management client, it is a program which implements a PTP management client according to IEEE1588 standard. The program reads from the standard input actions specified by name and management ID, sends them over the selected transport and prints any received replies. There are three actions supported:

- GET: Retrieves the specified information
- SET: updates the specified information
- CMD: (COMMAND) initiates the specified event

Command `help` can be used to get a list of supported actions and management IDs.
5.6.2.4 Ptp4l

PTP Boundary/Ordinary Clock, ptp4l is an implementation of the Precision Time Protocol according the IEEE1588 standard for Linux. It implements Boundary Clock and Ordinary Clock.

This is the main program used to synchronize the PHC of the ZedBoard with other clock over a LAN. The aim of the implementation of a PHC for Zynq-7000 is not synchronizing the Linux system time to a PTP hardware clock; on one hand, varying the time of the Linux system clock periodically can introduce unknown behaviors in the system and in the other hand it is not the goal of this development. So phc2sys, hwstamp_ctl and pmc programs will not be used.

Independently of the Linux system time, the PHC is synchronized. For instance, this clock could be useful for critical time dependent measures where each sample can be saved with its corresponding timestamp.

5.6.3 Debugging the User Space Program, GDB Server

The gdbserver is a computer program that makes it possible to remotely debug other programs. Running on the same system as the program to be debugged, it allows the GNU Debugger to connect from another system; that is, only the executable to be debugged needs to be resident on the target system, while the source code and a copy of the binary file to be debugged reside on the developer’s workstation. The connection can be either TCP or a serial line [26].

Eclipse CDT can be used for code development and remote debugging platform. Creating an executable project with cross GCC toolchain, eclipse will compile with the BuildRoot GCC for the target and can generate a debugging profile for testing directly in the target and a running profile to load it in the target and execute.

For setting up the cross debug platform, first the path of the BuildRoot cross-compiler has to be set in project settings as shown in Figure 204.
Other important thing to be done is to add the Linux library path as shown in Figure 205. In the previous sections it is seen that for building a program from sources the GNU Cross Compiler path is set in the environment variable called PATH, and the Linux libraries are set modifying the Makefile. For building from sources with Eclipse these things are predefined in the project options.
Also the path of the user libraries of the target has to be added as depicted in Figure 206.
Linux gdbserver support has to be added in BuildRoot. Open a Linux shell and place in the BuildRoot folder, and then run the Qt-based configurator. Typing the following command the configurator is launched, the new window is divided in three parts, navigate in the left pane to Toolchain, then in the right upper pane add Build gdb server for the target and Build gdb for the host as shown in Figure 207.

$ make xconfig
As explained in Chapter 5, close the configurator and re-building the `gdbserver` support will be included.

Finally, back to Eclipse, the debugger has to be configured creating a new launch configuration as shown in Figure 208.
The remote launcher has to be specified to been launch manually as shown in Figure 209.
In the main tab the program to debug has to be specified. The path of the binary of the debugger added by BuildRoot has to be included as depicted in Figure 210.

![Create, manage, and run configurations](image)

**Figure 210 Eclipse Program Development Cross Building Platform, Set the Path of the gdbserver**

The final part of the configuration is to set the TCP connection for the debugger as depicted in Figure 211.
For launching the gdbserver the following command has to be typed. The IP direction of the workstation where the eclipse is prepared for debugging the program, and the port number of the TCP connection as Figure 211 depicted, followed with the program to debug.

\$ gdbserver <IP.of.the.workstation>:<port> ./<program_to_debug>
Chapter 6: Conclusions and Future Work

6.1 Summary

The results of this work can be pointed out in two parts; the first one is the construction and integration of a custom Linux Operating System for the ZedBoard platform and the second one is the PTP IEEE1588 support added in the Ethernet driver layer for managing the specific hardware PTP-wise registers included in the Ethernet MAC layer and building from sources a user space program for implementing a PTP Boundary and Ordinary clock.

6.1.1 Customization, Construction and Integration of a Linux OS for ZedBoard Platform

Faint support for embedding Linux on ZedBoard were divulged in the beginning of this project; however the number of developers joining in Zynq-7000 technology has been increased widely, as a consequence, the amount of documentation currently is greater than in October 2012 when this work started.

For building a complete Linux distribution a wide range of concepts has to be acquired, 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.

The complete system has been integrated in BuildRoot for automating the building process and cross compiling; including BusyBox for basic user space utilities, a generation of a root filesystem image and the Linux kernel image. A complete set of instructions and figures have been created as a guide for understanding BuildRoot and its use for ZedBoard platform. Also, the top configuration of BuildRoot, named .config, is transformed and included to the default-configs folder/configs of the BuildRoot top level sources for future uploading to the BuildRoot developers to validate a new default configuration and including it in future releases.

For creating the first stage boot-loader it was necessary to use Xilinx development tools and the Linux boot-loader had been built with an external cross compiler different from BuildRoot’s.

Some isolated tests had been done previous the generation of all the different generated parts of the system; when finished building the boot.bin image it can be tested to confirm the bringing up of the system and the load process of the Linux loader. Whenever the rest of the files had been created the system had been fully tested.
6.1.2 Zynq-7000 Support for PTP IEEE1588 Standard

For adding full support for Precision Time Protocol IEEE1588 Standard, Linux driver development for accessing to the hardware capabilities implemented on the Ethernet MAC had been added, including PTP hardware clock creation and codification of its management methods.

The management methods were implemented within the Linux Ethernet driver written by Xilinx, taking advantage of very basically methods for accessing to some PTP physical registers of the Zynq-7000 that were already codified.

For the hardware adjustment of the frequency of the PTP clock seconds and nanoseconds registers of the Zynq-7000 the clock signal input of the times-tamping module has to be known. Referring to Xilinx Technical Reference Manual, [1] last update June 28 2013, the time-stamping module is clocked by CPU_1x signal with a frequency of 111MHz.

That frequency result indicates that for obtaining a 1 Hz-clock, each clock cycle the PTP nanoseconds register has to be incremented by 9 units of nanoseconds. In practice that value creates a clock that in 60 seconds the drift is 257.71 seconds approx. As a result, the clock frequency input of the time-stamping unit had been inferred obtaining that for achieving a 1Hz-clock each clock cycle the nanoseconds register will have to increment by 2.08 nanoseconds, that value creates a clock that in 60 seconds the drift is 797 microseconds.

One important problem of the Zynq-7000 times-tamping unit is the width of the frequency adjusting registers, only 24-bit width divided in three parts that only a Diophantine equation could be used to approximate an exact value.

6.2 Future Work

As the results of not achieving a sub-nanosecond synchronization with the linuxptp program, some alternative strategies are planned for obtaining more accuracy in the clock adjust. As linuxptp program has no documentation, the sources were still being studied for trying to modify the proportional and the integral parameters of the PI controller that governs the clock servo or even create other controller.

There are other Linux implementations of the IEEE1588, one remarkable is the PTPd project [27], for using this program some extra modifications have to be done for linking the user space program with the Ethernet driver level. Swapping to this alternative implementation is not dismissed.

The CPU_1x signal is going to be populated in an output pin of the Zynq-7000 for measuring with an oscilloscope the real frequency value of that signal. The results achieved in this work together with the information of the measured clock frequency are going to be reported to Xilinx for further support of the time-stamping unit.
Other alternative that have to be studied is for supporting IEEE1588 on Zynq-7000 is synthesizing an IP core in the PL to implement the Time Stamp Unit remaining the PTP frame recognition in the Ethernet core.
Chapter 7: Bibliography


Development of Embedded Linux Applications Using ZedBoard
