BORA

ARM Cortex-A9 + FPGA CPU Module

Ultra Line

Bora Embedded Linux Kit (*BELK*)

Quick Start Guide

DAVE Embedded Systems www.dave.eu info@dave.eu

<Page intentionally left blank>

Table of Contents

Index of Tables

Illustration Index

1 Preface

1.1 About this manual

This manual describes the BORA/BORAX Embedded Linux Kit (BELK) and serves as a quick guide for start working with the development kit.

1.2 Copyrights/Trademarks

Ethernet® is a registered trademark of XEROX Corporation.

All other products and trademarks mentioned in this manual are property of their respective owners.

All rights reserved. Specifications may change any time without notification.

1.3 Standards

DAVE Embedded Systems is certified to ISO 9001 standards.

1.4 Disclaimers

DAVE Embedded Systems does not assume any responsibility for availability, supply and support related to all products mentioned in this manual that are not strictly part of the BORA/BORAX CPU modules and the BORAEVB/BORAXEVB carrier boards.

BORA/BORAX CPU Modules are not designed for use in life support appliances, devices, or systems where malfunctioning of these products can reasonably be expected to result in personal injury. **DAVE Embedded Systems** customers who are using or selling these products for use in such applications do so at their own risk and agree to fully indemnify **DAVE Embedded Systems** for any damage resulting from such improper use or sale.

1.5 Warranty

BORA/BORAX SOMs and BORAEVB/BORAXEVB are guaranteed against defects in material and workmanship for the warranty period from the shipment date. During the warranty period, **DAVE Embedded Systems** will at its discretion decide to repair or replace defective products. Within the warranty period, the repair of products is free of charge provided that warranty conditions are observed.

The warranty does not apply to defects resulting from improper or inadequate maintenance or handling by the customer, unauthorized modification or misuse, operation outside of the product's specifications or improper installation or maintenance.

DAVE Embedded Systems will not be responsible for any defects or damages to other products not supplied by **DAVE Embedded Systems** that are caused by a faulty BORA/BORAX module or BORAEVB/BORAXEVB carrier boards.

1.6 Technical Support

We are committed to making our products easy to use and will help customers use our CPU modules in their systems.

Technical support is delivered through email for registered kits owners. Support requests can be sent to support-bora@dave.eu. Software upgrades are available for download in the restricted download area of **DAVE Embedded Systems** web site: http://www.dave.eu/reserved-area. An account is required to access this area.

Please refer to our Web site at http://www.dave.eu/products/som/xilinx/zynq-XC7Z010-XC7Z020_bora and http://www.dave.eu/products/som/xilinx/zynq-XC7Z015-XC7Z030_bora-xpress for the latest product documents, utilities, drivers, Product Change Notices, Board Support Packages, Application Notes, mechanical drawings and additional tools and software.

1.7 Related documents

Document	Location
DAVE Embedded Systems Developers Wiki	http://wiki.dave.eu/index.php/Main_Page
Zynq-7000 Technical Reference Manual	http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf
BORA main page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/Category:Bora
BORA Xpress main page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/Category:BoraX
BORA Hardware Manual	http://www.dave.eu/sites/default/files/files/bora-hm.pdf
BORAX Hardware description	http://wiki.dave.eu/index.php/Category:BoraX#Hardware
BORAEVB page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/BoraEVB
BORAXEVB page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/BoraXEVB
BORA integration guide on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/Integration_guide_%28Bora%29
Vivado Design Suite User Guide: Embedded Processor Hardware Design	http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug898-vivado-embedded-design.pdf

Document	Location
Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT)	http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/ug873-zynq-ctt.pdf
Zynq-7000 All Programmable SoC Software Developers Guide	http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf
Building Embedded Linux Systems By Karim Yaghmour.	This book covers all matters involved in developing software for embedded systems.
Training and Docs sections of Free Electrons website.	Brief but still exhaustive overview of the Linux and Embedded Linux world

Tab. 1: Related documents

1.8 Conventions, Abbreviations, Acronyms

Abbreviation	Definition
BTN	Button
BELK	BORA Embedded Linux Kit
EMAC	Ethernet Media Access Controller
GPI	General purpose input
GPIO	General purpose input and output
GPO	General purpose output
PCB	Printed circuit board
PL	Zynq Programmable Logic
PS	Zynq Processing System
PSU	Power supply unit
RTC	Real time clock
SOC	System-on-chip
SOM	System-on-module

Abbreviation	Definition
WDT	Watchdog

Tab. 2: Abbreviations and acronyms used in this manual

Revision History

Version	Date	Notes
1.0.0	July 2013	First release
1.0.1	September 2013	Minor fixes
1.0.2	September 2013	Added rfs info Added bit-to-bin script info Released with BELK 1.0.0
1.0.3	November 2013	Updated info for Vivado 2013.3 Minor fixes
1.0.4	November 2013	Minor fixes Added Vivado FAQ
1.0.5	November 2013	Fixed toolchain and rfs infos Added toolchain and rfs FAQs Added Appendixes section Minor fixes
1.0.6	April 2014	Minor fixes Added instructions for booting from SD Updated for BELK 2.0.0 release
1.0.7	July 2014	Fixed toolchain information
1.0.8	September 2015	Updated for BELK 2.2.0 release
1.0.9	March 2016	Updated for BELK 3.0.0 release

2 Introduction

2.1 BORA and BORA Xpress SOMs

BORA is the new top-class Dual Cortex-A9 + FPGA CPU module by **DAVE Embedded Systems**, based on the recent Xilinx Zynq XC7Z0xx application processor.

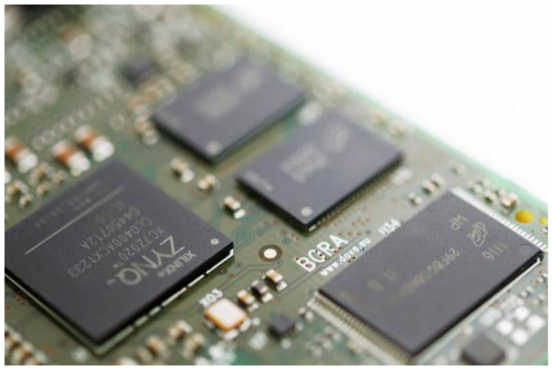


Fig. 1: BORA – Powered by Zynq processor

Thanks to BORA, customers are going to save time and resources by using a compact solution that **includes both a CPU and an FPGA**, avoiding complexities on the carrier PCB.

The use of this processor enables extensive system-level differentiation of new applications in many industry fields, where high performances and extremely compact form factor (85mm x 50mm) are key factors. Smarter system designs are made possible, following the trends in functionalities and interfaces of the new, state-of-the-art embedded products.

Fig. 2: BORA – Dual ARM Cortex A9 plus FPGA

BORA/BORAX offers great computational power, thanks to the rich set of peripherals, the Dual Cortex-A9 and the Artix-7 FPGA together with a large set of high-speed I/Os (up to 5GHz).

BORA/BORAX enables designers to create rugged products suitable for harsh mechanical and thermal environments, allowing the development of the most advanced and robust products.

Thanks to the tight integration between the ARM-based processing system and the on-chip programmable logic, designers are free to add virtually any peripheral or create custom accelerators that extend system performance and better match specific application requirements.

BORA/BORAX is designed and manufactured according to **DAVE Embedded Systems *Ultra* Line** specifications, in order to guarantee premium quality and technical value for customers who require top performances and flexibility. BORA/BORAX is suitable for high-end applications such as medical instrumentation, advanced communication systems, critical real-time operations and safety applications.

For further information on BORA/BORAX, please refer to the SOMs Hardware Manuals.

2.2 Embedded Linux

When we talk in general about Embedded Linux[1], we refer to an embedded system running Linux operating system. As the reader probably knows, Linux was first developed on the PC platform, based on the famous x86 architecture. Typical embedded systems using an operating system (O.S. for short), are equipped with much lighter software. Recent hardware advances made these systems so powerful that now they can run a complex O.S. such as Linux. This choice has several benefits:

- the developer can count on a reliable and efficient software, developed and maintained by a large community all over the world
- the software is open-source, so developers have access to the whole source code
- since Linux runs on many different platforms (x86, PowerPC, ARM, SuperH, MIPS etc.), applications are portable by definition
- there are a lot of open-source applications running on top of Linux that can easily be integrated in the embedded system
- last but not least, there are no license fees.

The typical Embedded Linux system is composed of:

- the bootloader – this software is run by the processor after exiting the reset state. It performs basic hardware initialization, retrieves the Linux kernel image (for example from a remote server via the TFTP protocol) and launches it by passing the proper arguments (command line and tags)
- the Linux kernel
- the root file system – this file system is mounted (which means "made available", "attached") by the kernel during the boot process on the root (“/”)

1 An exhaustive description of this topic is beyond the scope of this document. We recommend reading specific documents, eg Building Embedded Linux Systems By Karim Yaghmour.

directory

The typical developing environment for an Embedded Linux system is composed of a host machine and a target machine. The host is used by the developer to compile the code that will run on the target. In our case the target is obviously the BORA/BORAX module, while the host is assumed to be a PC running the Linux operating system. The Linux kernel running on the target can mount the root file system from different physical media. For example, during the software development, we strongly recommend using a directory exported via NFS by the host for this purpose; however, for system deployed to the field, the root file system is usually stored into a flash device.

2.3 Xilinx Zynq 7000 SOC

The Zynq™-7000 family SOCs integrate a feature-rich dual-core ARM® Cortex™-A9 based processing system (PS) and Xilinx programmable logic (PL) in a single device. The ARM Cortex-A9 CPUs are the heart of the PS, while the PL provides a rich architecture of user-configurable capabilities. The PS and PL can be tightly or loosely coupled using multiple interfaces and other signals that have a combined total of over 3,000 connections. This enables the designer to effectively integrate user-created hardware accelerators and other functions in the PL logic that are accessible to the processors and can also access memory resources in the PS. Zynq customers are able to differentiate their product in hardware by customizing their applications using PL.

In contrast with “typical” SOCs, where developers have to deal just with one main component (the CPU), Zynq SOC adds some complexity, since both the PS part and the PL part must be managed. Therefore, some knowledge of FPGAs and how they work would be valuable. However, the design process for Zynq-based systems is PS-centric: the processors in the PS always boot first, allowing a software centric approach for PL configuration. 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). This latter capability is analogous to the dynamic loading and unloading of software modules. The PL configuration data is referred to as a bitstream.

The addition of hardware programmability to the hardware and software interface imposes new requirements on design flows. Please refer to section 3.2.1 for an introduction to the Xilinx Zynq 7000 development tools (Vivado® Design Suite and Xilinx Software Development Kit).

2.4 BELK

BORA/BORAX Embedded Linux Kit (BELK for short) provides all the necessary components required to set up the developing environment for:

- configuring the system (PS and PL) at hardware level
- build the first-stage bootloader (FSBL)
- building the second stage bootloader (U-Boot)
- building and running Linux operating system on BORA/BORAX-based systems
- building Linux applications that will run on the target

DAVE Embedded Systems provides all the customization required (in particular at bootloader and Linux kernel levels) to enable customers use the standard Zynq-7000 development tools for building all the firmware/software components that will run on the target system.

2.4.1 Logical structure of BORA/BORAX Embedded Linux Kit (BELK)

To understand the structure of BELK, it is necessary to describe the basic organization of Xilinx Vivado Design Suite/Xilinx SDK and to recall briefly the recent history of development tools provided by Xilinx.

2.4.1.1 A little bit of history

At the time of this writing (October 2013) Xilinx is migrating from mature ISE 14.x Design Suite - that should be the last series of this suite - to the new Vivado environment. Both are composed by several programs and some of these are in common. From the general standpoint, the main difference between ISE and Vivado - even if ISE does support Zynq - is the latter has been expressively conceived to support newer SOC architectures such as Zynq, besides traditional FPGAs. Thus, adopting Vivado as the default environment for BELK would seem the natural choice. However, the migration process mentioned above has just begun and the majority of application notes and reference designs released by Xilinx still refers to ISE suite. Plus Vivado is still a little bit "green" and several bug fixes and improvements are introduced by every new release.

Since BORA was presented in 2013 and because this product addresses long longevity markets such as industrial and biomedical, **DAVE Embedded Systems** chose to build BELK upon Vivado, that certainly represents today the future of Xilinx development environments.

2.4.1.2 Basic structure of Vivado Design Suite and integration into BELK

Vivado/SDK[2] can be viewed as a collection of programs required to deal with all of the development aspects related to Xilinx components (software running on ARM cores, FPGA fabric verification and programming, power estimation etc.).

2 The Software Development Kit (SDK) is the Xilinx Integrated Design Environment for creating embedded applications on Zynq™-7000 All Programmable SoCs. SDK is the first application IDE to deliver true homogeneous and heterogeneous multi-processor design and debug, it is optionally included with the Vivado Design Suite or ISE Design Suite, or available as a separate free download for application developers.

These include strictly FPGA-related tools such as Floorplanner and pure-software development tools such as SDK. The ambitious objective is to provide a complete, user friendly, integrated environment that allows software developers to deal with FPGA development even if they are not familiar with this technology, by hiding a lot of its complexities[3]. As usual this ease of use comes at the expense of control and flexibility. This could not be acceptable in many cases where engineers need to control and customize many aspects of the project to implement what is required by system specifications. For this reason BELK has been built around Vivado but some deviations from the default development approach suggested by Xilinx have been introduced, in order to push the modularization and the maintainability of the projects to the maximum extent. More details about this can be found in chapter 3 on page 23.

3 Nevertheless, FPGA developers will find all the traditional tools that allow complete control of FPGA fabric.

2.4.2 Kit Contents

The following table lists the BELK components

Component	Description
	BORA/BORAX SOM
	BORAEVB/BORAXEVB Carrier board
	AC/DC Single Output Wall Mount adapter Output: +12V – 2.0 A
	MicroSDHC card with SD adapter and USB adapter

2.4.3 BELK Release Notes

2.4.3.1 Version 1.0.0

First official release

2.4.3.2 Version 1.1.0

Updates:

1. Switched to Vivado 2013.3
2. Added application note "AMP on BORA"

2.4.3.3 Version 2.0.0

Updates:

1. Added support for the BORAEVB carrier board
2. Updated drivers support (please refer to Section 2.4.3.8)

2.4.3.4 Version 2.1.0

Updates:

1. Fix and performance improvement on u-boot network interface
2. Fix SD card hotplug issue in Linux
3. First Yocto Daisy (1.6) BSP Release (please refer to section 3.4.6)

2.4.3.5 Version 2.2.0

Updates:

1. Switched to Vivado 2014.4
2. Added “board part” for the BORA SOM, which includes all the settings of the Zynq PS for BORA, allowing users to manage projects from the Vivado GUI (and not only using the CLI). Please refer to section 3.4.2.2
3. Added ConfigID (please refer to section 3.4.7)
4. Updated U-Boot and Linux versions

2.4.3.6 Version 3.0.0

Updates:

1. Added support for BORA Xpress SOM
2. Updated U-Boot and Linux versions

2.4.3.7 Known limitations

The following table reports the known limitations of the latest XELK version, which will be solved for the next releases of the development kit:

Issue	Description
ETH0 interface	On BORAEVB, there is a mistake in the connection of the center tap pins. They should be separated from one another and connected through separate 0.1µF common-mode capacitors to ground (for further details (eg: connection and selection of the magnetics), please refer to the Micrel KSZ9031RNX datasheet)
Yocto ubi-utils command line	Some Yocto ubi-utils command line are in non standard format. For ubimkvol and ubirmvol the UBI device must be the last argument (e.g. ubimkvol -m -N belk /dev/ubi0)
External DDR3 bank	The DDR3 SDRAM bank on the BORAEVB is not supported in this BELK version
ETH1 interface	The additional Gigabit Ethernet interface (ETH1) is not supported in this BELK version
RTC	Date/time retention is limited to about 4 hours

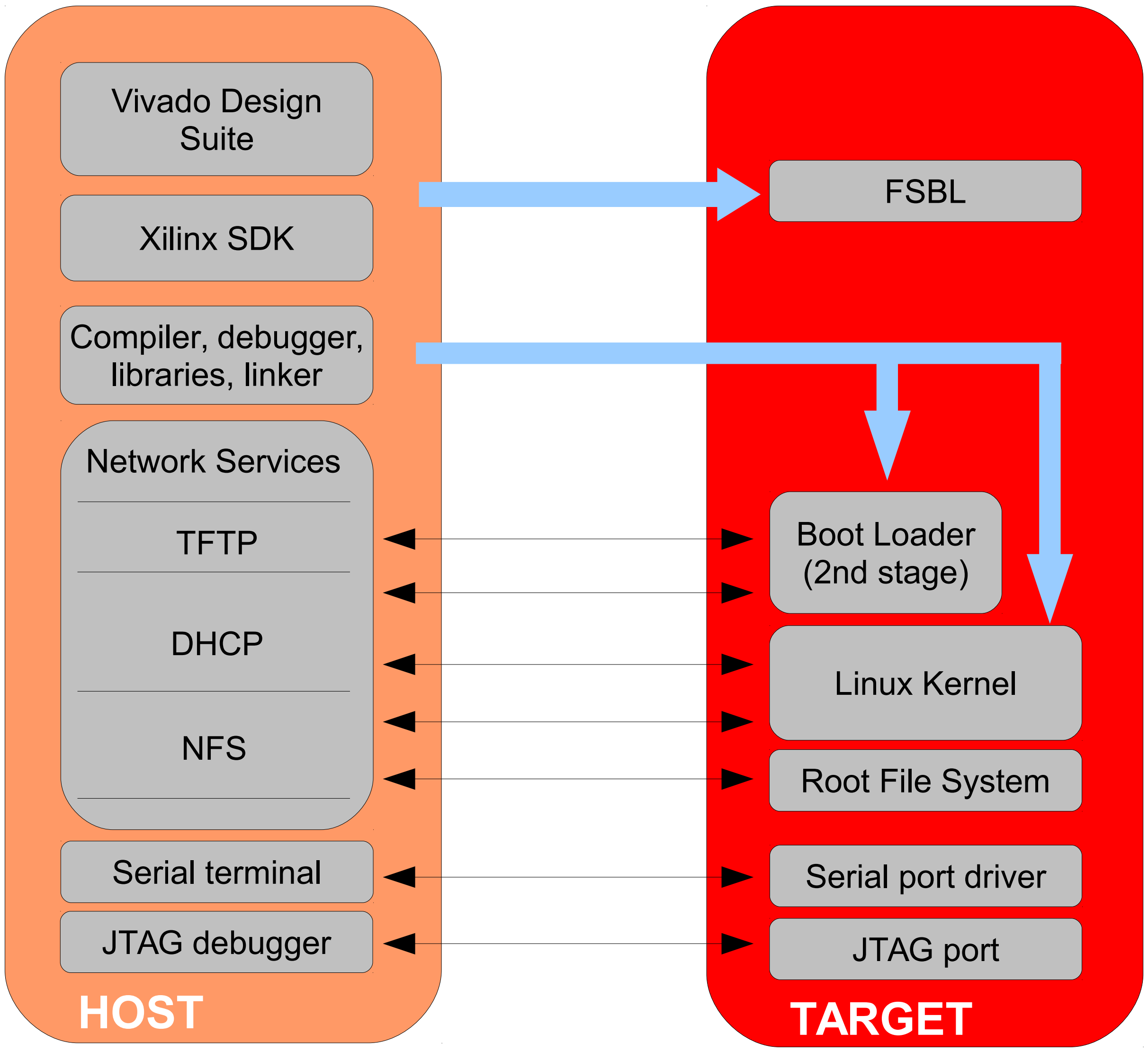
2.4.3.8 Releases history

BELK Version						
Release number	1.0.0	1.1.0	2.0.0	2.1.0	2.2.0	3.0.0
Status	Released	Released	Released	Released	Released	Released
Release date	July 2013	November 2013	April 2014	August 2014	September 2015	March 2016
Release notes	Version 1.0.0	Version 1.1.0	Version 2.0.0	Version 2.1.0	Version 2.2.0	Version 3.0.0
SOM PCB version	CS020313A	CS020313A	CS020313B	CS020313B	CS020313B	BORA:CS020313C BORAX: CS112714
Supported carrier boards	BORAEVB-Lite	BORAEVB-Lite	BORAEVB	BORAEVB	BORAEVB	BORAEVB BORAXEVB
U-Boot version	2013.04-belk-1.0.0	2013.04-belk-1.1.0	2013.04-belk-2.0.0	2013.04-belk-2.1.0	2014.07-belk-2.2.0	2014.07-belk-3.0.0
Linux version	3.9.0-bora-1.0.0	3.9.0-bora-1.1.0	3.9.0-bora-2.0.0	3.9.0-bora-2.1.0	3.17.0-bora-2.2.0	3.17.0-bora-3.0.0
Drivers	-	-	Gigabit Eth #0 UART NOR NAND SD/MMC USB Host/Device RTC CAN I2C	Gigabit Eth #0 UART NOR NAND SD/MMC USB Host/Device RTC CAN I2C	Gigabit Eth #0 UART NOR NAND SD/MMC USB Host/Device RTC CAN I2C ConfigID	Gigabit Eth #0 UART NOR NAND SD/MMC USB Host/Device RTC CAN I2C ConfigID
Vivado Version	2013.2	2013.3	2013.3	2013.3	2014.4	2014.4
Build System	-	-	-	Yocto Daisy (1.6)	Yocto Daisy (1.6)	Yocto Daisy (1.6)

3 Developing Environment

3.1 Introduction

The following figure shows the developing environment for an Embedded Linux system based on BORA or BORAX: it is composed of a host machine and a target machine.



In a typical environment, the host is used by the developer to (cross-)compile the code that is to run on the target. In our case the target is a SOM, while the host is assumed to be a PC running the Linux operating system, either in a physical installation or as a virtual machine. The bootloader running on the target can download the Linux kernel image through the network (TFTP), as well as the u-boot binary images (useful when an update of the bootloader is required). Moreover, the Linux kernel running on the target is able to mount the root file system from different physical media, for example from a directory exported via Network File System (NFS) by the host. This strategy (kernel image and RFS retrieved from the network) saves time during the development phase, since no flash reprogramming or removable storage (SD, usb pen drives, external disks) is required to test new versions or updates of the software components. In contrast with a typical embedded system, BORA/BORAX adds some complexity, due to the nature of the Zynq processor, which provides both a CPU core (PS, processing system) and a integrated FPGA (PL, programmable logic). This means that additional tools are required to manage this complexity. In particular, the Vivado® Design Suite and the Xilinx Software Development Kit are required for the hardware level configuration and for building the first stage boot loader (FSBL).

3.2 Software components

3.2.1 Xilinx Zynq-7000 development tools

3.2.1.1 Vivado® Design Suite

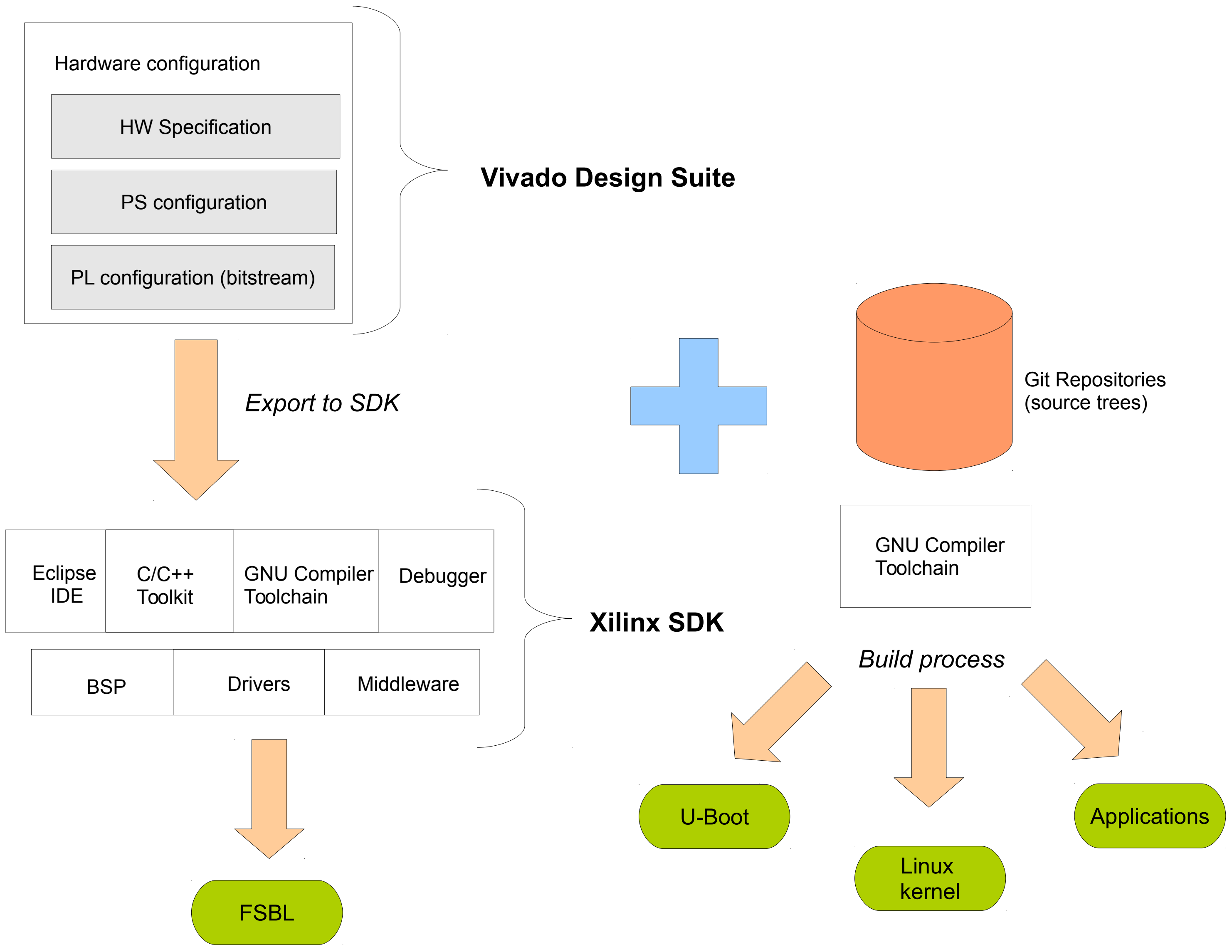
Xilinx provides the Vivado® Design Suite, a SoC-strength, IP-centric and system-centric, next generation comprehensive development environment for All Programmable SoCs. The Vivado® Design Suite is used to configure the PS for embedded processor hardware system development. Specification of the microprocessor, memory parameters, boot peripherals, MIO settings and the interconnection of these components, along with their respective detailed configuration, takes place in Vivado IP Integrator.

3.2.1.2 Xilinx Software Development Kit

The Software Development Kit (SDK) is the Xilinx Integrated Design Environment for creating embedded applications on Zynq™-7000 All Programmable SoCs. SDK is the first application IDE to deliver true homogeneous and heterogeneous multi-processor design and debug, it is optionally included with the Vivado Design Suite or ISE Design Suite, or available as a separate free download for application developers. It is based on Eclipse and CDT and it directly interfaces to the Vivado embedded hardware design environment. The SDK offers support for complete software design and debug flows including multi-processor and hardware/software debug capabilities, integrating editor, compilers, build tools, flash memory management, JTAG/GDB debug, custom libraries and device drivers.

3.2.1.3 Zynq application development flow

The following diagram describes the development flow for Zynq-based Linux applications:



3.2.2 Toolchain

With the term "toolchain" we refer to the set of programs that allow the building of a generic application. For applications built to run on the same platform as the tool chain, we use a native toolchain. On the contrary, for applications built to run on a target architecture different from the host architecture, we use a cross-toolchain. In this case all the tools involved in this process are lead by the “cross-” prefix. So we talk about cross-compiler, cross-toolchain and so on. The cross-toolchain used to build U-Boot, the Linux kernel and the applications is the GNU toolchain for the ARM architecture built for x86 hosts. In other words, the toolchain runs on x86 machines but generates binaries for ARM processors. As for all the software compliant to the GPL license, it is released in source code. Thus the first thing to do to set up the developing environment should be building the cross-toolchain. This is not a trivial task, it takes a lot of time and hard disk space. To avoid this tedious task, we suggest using a pre-built toolchain as explained in the following sections.

3.2.3 First stage bootloader (FSBL)

The first stage bootloader is loaded by the internal bootrom during the boot process. FSBL is run to set up the PS and load the second stage bootloader (U-Boot). The FSBL must be created using the Xilinx development tools and must be stored into the boot memory.

3.2.4 Second stage bootloader: U-Boot

U-Boot is a very powerful boot loader and it became the "de facto" standard on non-x86 embedded platforms. The main tasks performed by U-Boot are:

- hardware initialization
- starting a shell on the serial port allowing the user to interact with the system through the provided commands
- automatic execution of the boot script (if any)

After system power-up, U-Boot prints some information about itself and about the system it is running on. Once the bootstrap sequence is completed, the prompt is printed and U-Boot is ready to accept user's commands. U-Boot manages an environment space where several variables can be stored. These variables are extremely useful to permanently save system settings (such as ethernet MAC address) and to automate boot procedures. This environment is redundantly stored in two physical sectors of boot flash memory; the default variables set is hard-coded in the source code itself. User can modify these variables and add new ones in order to create his/her own custom set of configurations. The commands used to do that are `setenv` and `saveenv`. This process allows the user to easily set up the required configuration. Once U-Boot prompt is available, it is possible to print the whole environment by issuing the command `printenv`.

For further information on use of U-Boot, please refer to http://www.denx.de/wiki/view/DULG/UBoot

3.2.5 Kernel

Linux kernel for Zynq processors is maintained primarily by Xilinx, that constantly works in close cooperation with Linux community in order to push all the released drivers into mainstream kernel.

Kernels released within BELK derive directly from Xilinx Zynq kernels, with patches added by **DAVE Embedded Systems** to support the BORA hardware platform.

3.2.6 Target root file system

The Linux kernel running on the target needs to mount a root file system. Building a root file system from scratch is definitively a complex task because several well known directories must be created and populated with a lot of files that must follow some standard rules. Again we will use pre-packaged root file systems that make this task much easier. Please note that using a pre-packaged root file system can lead to conflicts with the application binaries obtained using the pre-built cross-toolchain: as a general rule, dynamically linking an application against libraries built with a different toolchain can cause malfunctioning.

3.3 Build system

3.3.1 Introduction

A build system is a set of tools, source trees, Makefiles, patches, configuration files and scripts that makes it easy to generate all the components of a complete embedded Linux system. A build system, once properly set up, automates the configuration and cross-compilation processes, generating all the required targets (userspace packages (libraries, programs), the kernel, the bootloader and root filesystem images) depending on the configuration. In particular, using an integrated build system prevents from problems caused by misaligned toolchains, since a unique toolchain is used to build all the software components, including the customer application. Some well known structured build systems are the following:

- OpenEmbedded (http://wiki.openembedded.net/index.php/Main_Page)
- Yocto (https://www.yoctoproject.org/)
- Buildroot (http://buildroot.uclibc.org)

BELK does not provide a fully structured build system, since various heterogeneous tools are required to build the software components for the BORA/BORAX SOM. In particular, the Xilinx Zynq 7000 development tools are required to configure the system and build the FSBL, while the standard GNU tools are required to build U-Boot, kernel and user-space applications.

In the following section, we will refer to the system running the Xilinx tools (that can be either a Microsoft Windows machine or a GNU/Linux machine) as the “Zynq development server”, and to the machine running the GNU/Linux tools as the “Linux development server”.

3.3.2 Setting up the Zynq development server environment

The following software packages must be installed on the Zynq development server:

- Vivado® Design Suite version **2014.4**
- Xilinx Software Development kit
- Python 2.7.x (C:\Python27 must be the installation directory on Windows)
- A Git tool (e.g. for Windows: MsysGit (http://msysgit.github.io/))

The Zynq 7000 development tools can be downloaded from the Xilinx website: http://www.xilinx.com/support/download/index.htm in the WebPACK™ Edition, which is a free version that provides instant access to the fundamental Vivado features and functionality at no cost. For the hardware requirements of the PC, please refer to http://www.xilinx.com/design-tools/vivado/memory.htm#zynq-7000.

The Git tool is used to download the BORA/BORAX project files for Vivado from our public git repositories, as described in the next section.

N.B. Sometimes the download of the Vivado 2014.4 full package fails because of some download system malfunctioning, but the problem is barely noticeable, except by performing the MD5 check of the downloaded file. In case of problems, we suggest using the Multi-File Download (available on the same web page), that splits the full package in a collection of smaller files. If you use the Multi-File Download to get the "Vivado (No SDK)" package, you must also download the "Software Development Kit - 2014.4" package.

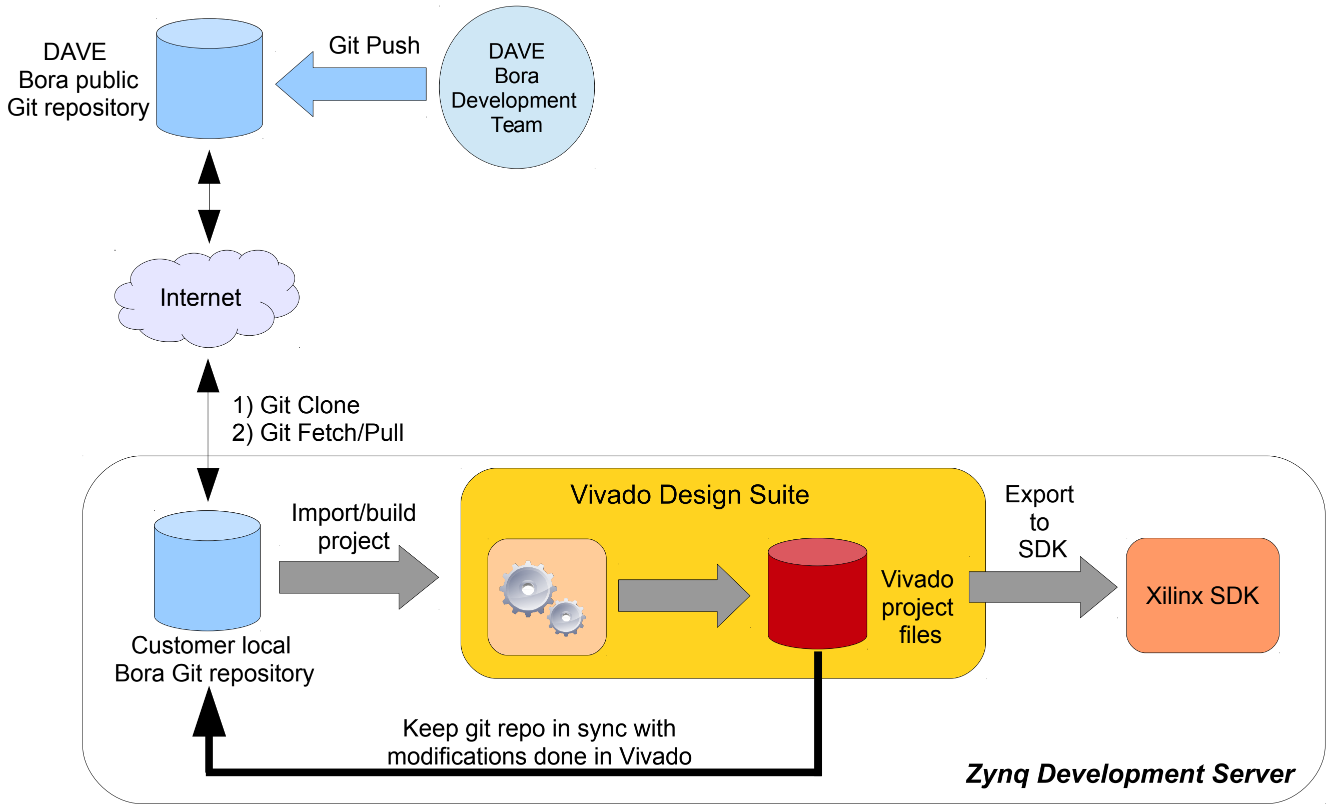
3.3.2.1 How BORA/BORAX project files are managed

Since a Vivado project for the Zynq device is composed by a lot of files (including temporary and GUI-managed files), providing the BORA/BORAX project files with the BELK is not considered as an efficient and user-friendly solution, for two main reasons:

- we should provide a compressed archive for each version (updates, new features, bug fixes) of the project, wasting storage space and download bandwidth, introducing redundancy and complicating file management
- there is no version control, which means that it's not possible to track changes to the project files, making development and release management complicated and error-prone

The best solution to these problems is that we create and maintain a Git repository to store and track only the main files of the Vivado project. Therefore, the developer can clone the repository and keep it in sync with our modifications.

The following diagram shows how the solution works:



BELK also provides the tools for keeping the files modified within the Vivado tools in sync with the local Git repository: with this solution, all the modifications are tracked and developers can take advantage of all the benefits of using git as a version control system.

The public git repository for BORA/Zynq project files is

git@git.dave.eu:dave/bora/bora.git

Please note that BELK distribution provides the git archive of the **.git** directory of the repository, so the user can immediately get access to the development tree (please refer to 3.3.3.7).

3.3.3 Setting up the Linux development server environment

During development, user needs to interact with the target system. This section describes the tools that must be installed and configured on the Linux host system for this purpose.

3.3.3.1 TFTP Server

One of the most useful features of a bootloader during development is the capability to download the Linux kernel from the network. This saves a lot of time because developer doesn't have to program the image in flash every time he/she modifies it. U-Boot implements the TFTP protocol (see the tftp command), so the host system must be configured to enable the TFTP service. Installation and configuration of a TFTP server depends on the host Linux distribution.

3.3.3.2 NFS Server

One of the most important components of a Linux system is the root file system. A good development root file system provides the developer with all the useful tools that can help him/her on his/her work. Such a root file system can become very big in size, so it's hard to store it in flash memory. User could split the file system in different parts, mounting them from different media (flash, network, USB...). But the most convenient thing is to mount the whole root file system from the network, allowing the host system and the target to share the same files. In this way, the developer can quickly modify the root file system, even “on the fly” (meaning that the file system can be modified while the system is running). The most common way to setup a system like the one described is through NFS (installation and configuration depends on the host Linux distribution).

3.3.3.3 Pre-built toolchain

To start developing software for the BORA platform, users need a proper toolchain, which can be pre-built or built-from-scratch. Building a toolchain from scratch is not a trivial task (though using a recent build system is easier than in the past), so the recommended approach consists in using a pre-built toolchain.

The toolchain used as a reference for BELK is the toolchain provided with the Xilinx SDK (usually installed into `/opt/Xilinx/SDK/<Vivado_version>/gnu/arm/lin/bin`).

Once the toolchain is installed, create a a bash script (`env.sh`) containing the following lines:

```
export PATH=<path_to_toolchain>:$PATH

export ARCH=arm

export CROSS_COMPILE=<toolchain_prefix>
```

For example, for the Vivado 2014.4 release, the variables are the following:

```
export PATH=/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin:
$PATH

export ARCH=arm

export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Use the following command to set up the environment shell variables required during the building procedure:

```
source env.sh
```

3.3.3.4 Pre-built root file system

Linux needs a root file system: a root file system must contain everything needed to support the Linux system (applications, settings, data, ..). The root file system is the file system that is contained on the same partition on which the root directory is located. The Linux kernel, at the end of its startup stage, mounts the root file system on the configured root device and finally launches the `/sbin/init`, the first user space process and "father" of all the other processes.

For more information on the Linux filesystem, please refer to http://www.thegeekstuff.com/2010/09/linux-file-system-structure/

BELK provides a pre-built root file system, that can be used during the evaluation/development phase, since it provides a rich set of packages for working with the BORA platform. Since this pre-built root file system is not generated using the same cross-toolchain used for building the BELK software components, we recommend to choose one of the following options:

- if a native compiler is available on the root file system, go for native compilation instead of cross-compilation
- when you cross-compile, rely on static linking and avoid dynamic linking against the root file system libraries
- build your application using the same cross-toolchain (when available) used for building the root file system

Please refer to Sections 3.2.6 and 4.6 for further details.

3.3.3.5 U-Boot and Linux git repositories

BELK source trees for U-Boot and Linux kernel are provided as git repositories.

Component	Git Remote
U-Boot	git@git.dave.eu:dave/bora/u-boot-xlnx.git
Linux	git@git.dave.eu:dave/bora/linux-xlnx.git

This means that these components can be kept in sync and up to date with **DAVE Embedded Systems**' repositories.

When the git account is enabled (please refer to section 3.3.3.6), the developer can:

- clone the repository with the `git clone <git_remote_repository>` command
- synchronize a source tree entering the repository directory and

launching the `git fetch origin` command

Please note that git fetch doesn't merge the commits on the current branch. To do that, the developer should run the `git merge` command or replace the "fetch-merge" process with a single `git pull` command. Please note that the recommended method is the "fetch-merge" process. For further information on Git, please refer to the official Git Documentation (http://git-scm.com/documentation).

For instructions on how to update the repositories from BELK 2.1.0 to BELK 3.0.0, please refer to section 3.4.1.

3.3.3.6 RSA key generation

For getting access to the Git repositories, a ssh key is required. Please follow the procedure reported below to generate the RSA ssh key (we assume that the ssh package and the required tools are installed on the Linux development server):

- select your username (ad es. username@myhost.com)
- start a shell session on the Linux host
- enter the `.ssh` subdirectory into your home directory:

```
cd ~/.ssh/
```

- launch the following command:

```
ssh-keygen -t rsa -C "username@myhost.com"
```

- this command creates the files `~/.ssh/username@myhost.com` ('''private key''') and `~/.ssh/username@myhost.com.pub` ('''public key''')
- edit your `~/.ssh/config` adding the following lines:

```
Host git.dave.eu
     User git
     Hostname git.dave.eu
     PreferredAuthentications publickey
     IdentityFile ~/.ssh/username@myhost.com
```

Please send the public key file to the following email support addresses support-bora@dave.eu

with the request for the creation of a new public git account associated to your username. The support team will enable the account and send you a confirmation as soon as possible.

3.3.3.7 Using the pre-packaged git archive

BELK distribution provides (on request) the git archive of the **.git** directory of the repositories, so the user can immediately get access to the development trees. Uncompressing the archive enables access to the hidden .git directories of the repositories; to get the source files, the developer must enter the project directory (eg: `bora/linux-xlnx.git`) and lauch the `git checkout <bora current branch>` command, like in the following examples:

For u-boot: `git checkout bora`

For linux: `git checkout bora`

Once these steps are completed, don't forget to update the repositories, as described in 3.3.3.5.

3.4 Working with BELK

The following sections describe how to perform the most common tasks for building the software components for a BORA-based embedded system.

3.4.1 Update from BELK 2.1.0 to BELK 3.0.0

To update the repositories from the previous BELK 2.1.0 to BELK 3.0.0, the following commands should be used:

```
git fetch origin

git checkout -b <new_branch_name> origin/bora
```

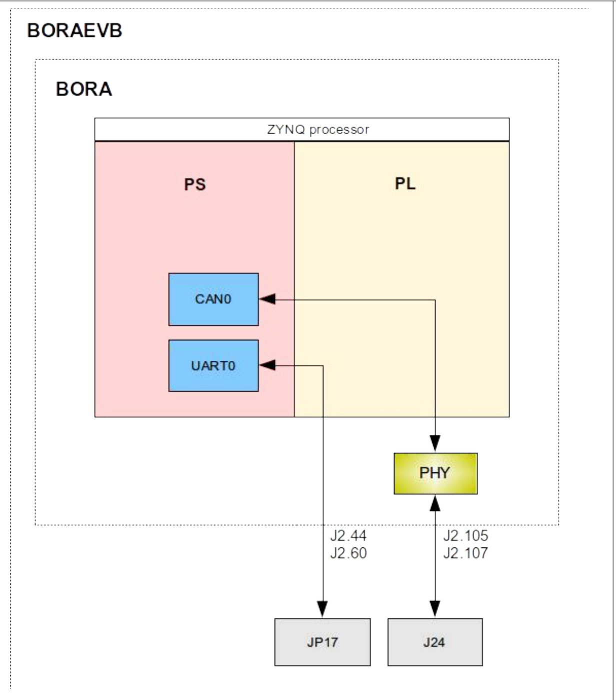
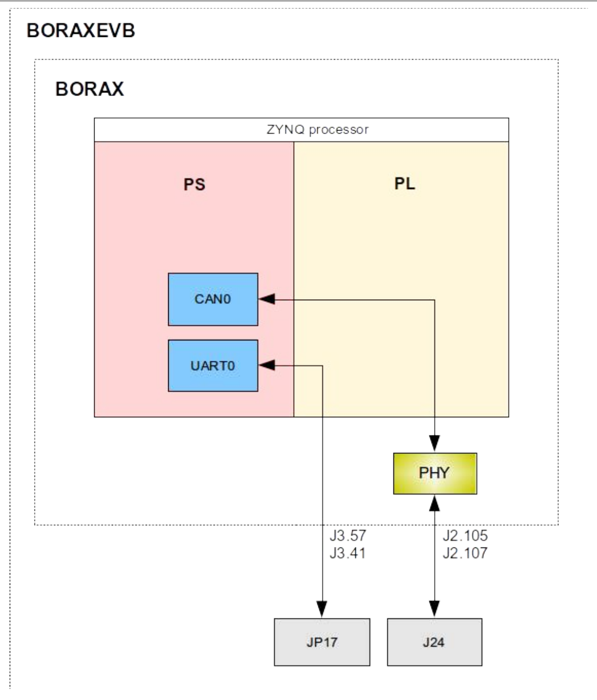
3.4.2 Creating and building a Zynq project for BORA/BORAX

BELK provides an example Vivado project for BORA/BORAX boards. This project allows to:

- generate the FSBL binary image
- generate the bitstream of a simple PL design used to route PS' CAN0 and UART0 signals through EMIO (see also the following pictures)

This article describes how two build this project. Two procedures are described, the former is command line based while the latter is GUI based.

The project is stored is a git repository, as described in section 3.3.2.1. It is assumed that the Zynq development environment has been set up properly (see section 3.3.2 for more details).

 BORAEVB BORA ZYNQ processor PS PL CAN0 UART0 PHY J2.44 J2.60 J2.105 J2.107 JP17 J24	 BORAXEVB BORAX ZYNQ processor PS PL CAN0 UART0 PHY J3.57 J3.41 J2.105 J2.107 JP17 J24
Fig. 3: Block diagram of BORA example project	**Fig. 4**: Block diagram of BORAX example project

3.4.2.1 Command line based procedure

The following procedure is detailed for BORA SOM. For BORAX, please replace:

- bora_FSBL with borax_FSBL
- bora_wrapper_hw_platform_0 with borax_wrapper_hw_platform_0
- bora.sdk with borax.sdk
- bora_wrapper.bit with borax_wrapper.bit

The procedure is the following:

- start the Zynq development server and login into the system
- assuming that a local repository has not been created, clone the remote BORA git repository:

```
git clone git@git.dave.eu:dave/bora/bora.git
```

- copy the `<bora_repo>/boards/board_parts/zynq/BORA` and `<bora_repo>/boards/board_parts/zynq/BORAX` directories to `<vivado_2014.4_install_dir>/data/boards/board_parts/zynq/` :

  ```
  cd <bora_repo>
  ```

  ```
  sudo cp -r boards/board_parts/zynq/BORA /opt/Xilinx/Vivado/2014.4/data/boards/board_parts/zynq/
  ```

  ```
  sudo cp -r boards/board_parts/zynq/BORAX /opt/Xilinx/Vivado/2014.4/data/boards/board_parts/zynq/
  ```

- enter the git directory and launch the following command

  ```
  export PROJ_DIR=$(pwd)/../bora-build-YYYYMMDD-nobk
  ```

- launch the Vivado Design Suite with the following commands:

  ```
  . /opt/Xilinx/Vivado/2014.4/settings64.sh
  ```
  [4]

  ```
  vivado -mode tcl -source build_project.tcl -notrace -tclargs "-bitstream"
  ```
  [5]

- the `build_project` script allows the user to select BORA or BORAX target
- at the end of the bitstream build process, the `build_project` script allows to automatically export hardware and **lauch SDK** to build the FSBL
- once the Xilinx SDK is ready, perform the following operations from the GUI:
  - Click on **File -> New -> Application Project**
  - Select the **Project Name**: *bora_FSBL*
  - Click **Next**

4 In a 32 bit system, Vivado settings are configured with the following command `/opt/Xilinx/Vivado/2014.4/settings32.sh`

5 Passing the -tclargs "-bitstream" parameters allows for automatic building of the FPGA bitstream.

  - Select **Template**: *Zynq FSBL*
  - Click on **Finish**
  - Apply the patch, right-clicking on bora_FSBL in Project Explorer and then clicking on **Team -> Apply Patch..**
  - From **Browse...** open the file `<bora_repo>/patch/belk-sd-boot.patch`
  - Click **Next**
  - Select **Apply the patch to the selected file, folder or project***:* and select `main.c` from **bora_FSBL -> src**
  - Click **Next**
  - Check that the patch is correctly applied to the source code and click on **Finish**
- with the same procedure, apply the patches to fix DDR3 CKE deassertion time (see also http://www.xilinx.com/support/answers/65145.html):
  - apply `<bora_repo>/patch/AR65145_ps7_init_c.patch` on `ps7_init.c` under `bora_wrapper_hw_platform_0`
  - apply `<bora_repo>/patch/AR65145_ps7_init_tcl.patch` on `ps7_init.tcl` under `bora_wrapper_hw_platform_0`
- the FSBL (ELF file) is built automatically
- create the binary from the FSBL ELF chosing one of the following options:
  - (this step is board dependent) manually launch the command: `arm-xilinx-eabi-objcopy -v -O binary $PROJ_DIR/bora.sdk/SDK/SDK_Export/bora_FSBL/Debug/bora_FSBL.elf $PROJ_DIR/bora.sdk/SDK/SDK_Export/bora_FSBL/Debug/bora_FSBL.bin`
  - (this step is board dependent) configure the automatic binary generation on project build. In Project Explorer, right-click on "bora_FSBL" project and select **C/C++ Build Settings** and add the

command `arm-xilinx-eabi-objcopy -v -O binary ${ProjName}.elf ${ProjName}.bin` on **Post-build steps**

- create the BOOT.bin image (single file including FSBL, FPGA and U-boot for uSD boot:
  - select the bora_FSBL project in **Project Explorer**
  - click on **Xilinx Tools -> Create Zynq Boot Image**
- if the project is correctly configured, the tool builds automatically all the component listed in the form, so just add U-Boot to the list.
- otherwise, select **Create new BIF** file and set the output path and in **Boot image partitions** add the following files:
  - *bora_FSBL.elf*, which can be found in the project Debug directory. N.B. check that the Partition Type for FSBL is bootloader
  - *bora_wrapper.bit*, which is the bitstream generated by the Vivado project (Partition Type must be Datafile)
  - *u-boot.elf*, which is the compiled U-Boot with .elf extension (Partition Type must be Datafile)
- in **Output path**, select the path for the BOOT.bin file

3.4.2.2 GUI based procedure

The following procedure is detailed for BORA SOM. For BORAX, please replace:

- bora_wrapper.v with borax_wrapper.v
- bora_pinout.xdc with borax_pinout.xdc
- bora_timings.xdc with borax_timings.xdc
- bora_FSBL with borax_FSBL
- bora.sdk with borax.sdk
- bora_FSBL.elf with borax_FSBL.elf
- bora_wrapper.bit with borax_wrapper.bit

The procedure is the following:

- start the Zynq development server and login into the system
- assuming that a local repository has not been created, clone the remote BORA git repository:

  ```
  git clone git@git.dave.eu:dave/bora/bora.git
  ```

- copy the `<bora_repo>/boards/board_parts/zynq/BORA` and `<bora_repo>/boards/board_parts/zynq/BORAX` directories to `<vivado_2014.4_install_dir>/data/boards/board_parts/zynq/` :

  ```
  cd <bora_repo>
  sudo cp -r boards/board_parts/zynq/BORA /opt/Xilinx/Vivado/2014.4/data/boards/board_parts/zynq/
  sudo cp -r boards/board_parts/zynq/BORAX /opt/Xilinx/Vivado/2014.4/data/boards/board_parts/zynq/
  ```

- launch the Vivado Design Suite GUI with the following commands:

  ```
  . /opt/Xilinx/Vivado/2014.4/settings64.sh
  vivado
  ```

- from the **start page** click on **Create New Project**
- click **Next**
- select the directory build project, insert the name of the project (**Project Name**) and click **Next**
- select **RTL Project**, enable **Do not specify sources at this time** and click **Next**
- on the **Default Part** form, click on the **Boards** button to filter the available boards. Select BORA or BORAX depending on target SOM and click **Next**
- check the summary page and click **Finish**
- in the Vivado GUI click on **Create Block Design** from

the **Flow Navigator**

- insert *bora* (or `borax`) as **Design name** and click OK
- this creates a new block design. From the Diagram tab, add a new IP:
  - click the **Add IP** side button, or
  - click **Add IP** on the upper suggestions bar
- double click on **ZYNQ7 Processing System**
- this adds the IP that models the PL component of Zynq. Launch **Run Block Automation** from the upper suggestions bar
- check that **Apply Board Preset** is selected and click OK
- this applies the default settings for BORA/BORAX and creates the I/O ports for the DDR and MIO pins and for the UART_0 and CAN_0 interfaces
- manually connect the **FCLK_CLK0** signal to **M_AXI_GP0_ACLK** and save the block design
- from the sources tab, select the BORA/BORAX block design (`bora.bd` for BORA, `borax.bd` for BORAX) as **Design Sources** and from the context menu select **Create HDL Wrapper**
- on the next window, select **Copy generated wrapper** to allow user edits and click OK
- this creates the Verilog file (`bora_wrapper.v` for BORA, `borax_wrapper.v` for BORAX). If this file is not automatically included in the project, add it using the **Add sources** option
  - select **Add or create design sources** and click **Next**
  - select the `bora_wrapper.v` file from the `<prj_name>.srcs/sources_1/bd/bora/hdl/` directory
- select **Add sources** and click on **Add or create constraints**
- select the `bora_pinout.xdc` and `bora_timings.xdc`

files from the `constr` directory of the BORA repository

- check that the option **Copy constraints files into project** is enabled
- create the synthesis, implementation and bitstream clicking **Generate Bitstream** from the **Flow Navigator** and wait the completion of the operation
- once completed, select **Open Implemented Design**
- create the binary bitstream running the tcl script provided with the BORA repository. Launch **Tools -> Run Tcl Script**
- select the `generate_binary_bitstream.tcl` file from the scripts directory from the BORA repository
- select **File -> Export -> Export Hardware**
- on the next window, enable **Include Bitstream** and click OK
- now launch the SDK session to generate the FSBL, clicking on **File -> Launch SDK**
- once the Xilinx SDK is ready, perform the following operations from the GUI:
  - Click on **File -> New -> Application Project**
  - Select the **Project Name**: *bora_FSBL*
  - Click **Next**
  - Select **Template**: *Zynq FSBL*
  - Click on **Finish**
  - Apply the patch, right-clicking on bora_FSBL in Project Explorer and then clicking on **Team -> Apply Patch..**
  - From **Browse…** open the file `<bora_repo>/patch/belk-sd-boot.patch`
  - Click **Next**
  - Select **Apply the patch to the selected file, folder or project***:* and select `main.c` from **bora_FSBL -> src**
  - Click **Next**

  - Check that the patch is correctly applied to the source code and click on **Finish**
- with the same procedure, apply the patches to fix DDR3 CKE deassertion time (see also http://www.xilinx.com/support/answers/65145.html):
  - apply `<bora_repo>/patch/AR65145_ps7_init_c.patch` on `ps7_init.c` under `bora_wrapper_hw_platform_0`
  - apply `<bora_repo>/patch/AR65145_ps7_init_tcl.patch` on `ps7_init.tcl` under `bora_wrapper_hw_platform_0`
- the FSBL (ELF file) is built automatically
- create the binary from the FSBL ELF chosing one of the following options:
  - (this step is board dependent) manually launch the command: `arm-xilinx-eabi-objcopy -v -O binary $PROJ_DIR/bora.sdk/SDK/SDK_Export/bora_FSBL/Debug/bora_FSBL.elf $PROJ_DIR/bora.sdk/SDK/SDK_Export/bora_FSBL/Debug/bora_FSBL.bin`
  - (this step is board dependent) configure the automatic binary generation on project build. In Project Explorer, right-click on "bora_FSBL" project and select **C/C++ Build Settings** and add the command `arm-xilinx-eabi-objcopy -v -O binary ${ProjName}.elf ${ProjName}.bin` on **Post-build steps**
- create the BOOT.bin image (single file including FSBL, FPGA and U-boot for uSD boot:
  - select the bora_FSBL project in **Project Explorer**
  - click on **Xilinx Tools -> Create Zynq Boot Image**
- if the project is correctly configured, the tool builds automatically all the component listed in the form, so just add U-Boot to the list.
- otherwise, select **Create new BIF** file and set the output path and in **Boot image partitions** add the following files:
  - *bora_FSBL.elf*, which can be found in the project Debug directory. N.B. check that the Partition Type for FSBL is

bootloader

    - *bora_wrapper.bit*, which is the bitstream generated by the Vivado project (Partition Type must be Datafile)
    - *u-boot.elf*, which is the compiled U-Boot with .elf extension (Partition Type must be Datafile)
- in **Output path**, select the path for the BOOT.bin file

3.4.3 Building U-Boot

It is assumed that the Zynq development environment has been set up properly (see section 3.3.2 for more details).

- start the Linux development server and login into the system
- assuming that a local repository has not been created, clone the remote U-Boot git repository (the "-b" option is used to automatically checkout the current branch):

```
git clone
git@git.dave.eu:dave/bora/u-boot-xlnx.git -b
bora
```

- Setup the server environment (please refer to section 3.3.3.3)
- enter the source tree directory and run the following commands:

```
make bora_qspi

make bora_mmc

make bora_noflash
```

The available targets are the following:

Target	Description
bora_qspi	this target is available for BORA only. U-Boot is built to use NOR flashed based environment
bora_mmc	this target is available for BORA only. U-Boot environment is placed in SD/MMC card

Target	Description
bora_noflash	this target is available for BORA and BORAX. U-Boot environment is placed in SD/MMC card. NOR flash and NAND flash are disabled

- build U-Boot by entering the `make` command. This will generate U-Boot binary images

Subsequent builds just require ''make'' command, without targets, to update the binary images. Once the build process is complete, the binary images can be copied to the tftp root directory (eg. `/srv/tftp/belk/`) with the following command:

```
cp u-boot.bin /srv/tftp/belk/
```

3.4.4 Building Linux kernel

It is assumed that the Zynq development environment has been set up properly (see section 3.3.2 for more details).

- start the Linux development server and login into the system
- assuming that a local repository has not been created, clone the remote Linux git repository (the "-b" option is used to automatically checkout the current branch):

```
git clone
git@git.dave.eu:dave/bora/linux-xlnx.git -b
bora
```

- setup the server environment (please refer to section 3.3.3.3)
- enter the source tree directory and run the following commands:

```
make bora_defconfig

make UIMAGE_LOADADDR=0x8000 uImage bora.dtb
```

The former command selects the default BORA/BORAX configuration, while the latter builds the kernel binary image with the required u-boot header and the kernel device tree. Please note that the `mkimage` tool is required for building the `uImage` binary. This tool must be installed on the Linux development server (please refer to the package manager of the Linux distribution).

Default linux kernel configuration can be changed by using the standard `menuconfig`, `xconfig`, or `gconfig` make target. Subsequent builds just require `uImage` make target to update the binary image. Once the build process is complete, the kernel binary image is stored into the `arch/arm/boot/uImage` file. Both this file and the kernel device tree can be copied to the tftp root directory (eg. `/srv/tftp/belk/`) with the following commands:

```
cp arch/arm/boot/uImage /srv/tftp/belk/
cp arch/arm/boot/dts/bora.dtb /srv/tftp/belk/
```

3.4.5 Booting the system via NFS

Assuming that:

- the CPU module is booting with a working FSBL and U-Boot image (either from flash NOR SPI or flash NAND or MicroSD card)
- a kernel image and a device tree binary are available and ready to be downloaded through tftp
- the root file system has been uncompressed into a nfs share

the system can boot using the `net_nfs`[6] configuration, as described in

http://wiki.dave.eu/index.php/Booting_Linux_Kernel#Configuration_net_nfs

6 The net_nfs configuration, besides setting the system for booting from the network, triggers a command (program_fpga) which loads the FPGA binary from TFTP and programs the bitstream

3.4.6 Building the software components with Yocto

The build process creates an entire Linux distribution from source. The build process can be summarized as follows:

- Make sure that all the prerequisites are met
- Initialize the build environment, as described in 3.4.6.2.
- Optionally ensure the conf/local.conf configuration file, which is found in the Build Directory, is set up how you want it. This file defines many aspects of the build environment including the target machine architecture through the MACHINE variable, the development machine's processor use through the BB_NUMBER_THREADS and PARALLEL_MAKE variables, and a centralized tarball download directory through the DL_DIR variable.
- Build the image using the bitbake command. If you want information on BitBake, see the BitBake User Manual.

3.4.6.1 Prerequisites

The following prerequisites are required and only need to be done once.

Some generic development tools are required on the host Linux machine:

- git
- curl
- build-essential
- diffstat
- texinfo
- gawk
- chrpath
- ia32-libs (if the host machine is running a 64-bit OS)

- python-m2crypto

These packages can be installed with the following command:

```
sudo apt-get install gawk wget git-core diffstat unzip
texinfo gcc-multilib build-essential chrpath socat
libsdl1.2-dev xterm
```

It is also recommended to switch the system shell from Ubuntu's standard dash to more universal bash:

```
:~$ sudo dpkg-reconfigure dash
```

3.4.6.2 Initializing the build environment

In the BELK, we have simplified the Yocto initialization phase, relying on the repo tool and on a BORA/BORAX bsp git repository, so that the initialization can be completed with a few commands as reported below:

```
:~$ mkdir belk && cd belk

:~/belk$ curl
http://commondatastorage.googleapis.com/git-repo-downl
oads/repo > repo

:~/belk$ chmod a+x repo

:~/belk$ ./repo init -u
git@git.dave.eu:dave/bora/bora-bsp.git -b bora

:~/belk$ ./repo sync
```

3.4.6.3 Build the Yocto image

Please note that even building the basic root file system requires a few hours to complete the process on a mid-hi range desktop PC (4-6 cores, 8-12 GiB RAM), also depending on the Internet connection speed (all source are fetched from the network). Nearly 20GiB of disk space is required for the build.

Once completed the initialization phase, developers can launch the Yocto image build process with the following commands:

```
:~$ cd ~/belk

:~/belk$ source bora-bsp-init-env.sh build

:~/belk/build$ bitbake base-rootfs-image
```

Please note that even the basic root file system requires a few hours to build on a mid-hi range desktop (4-6 cores, 8-12 GiB RAM) also depending on your Internet connection speed (all sources are fetched from the network). Nearly 20GiB of disk space is required for the build.

The resulting files (kernel, devicetree and u-boot binaries, plus .tar.gz root file system) will then be available inside `build/tmp/deploy/images/bora`.

3.4.6.4 Build additional packages

To build additional packages the user must first enter the directory where the bora-bsp-init-env.sh is placed and source it:

```
:~$ cd ~/belk
```

```
:~/belk$ source bora-bsp-init-env.sh build
```

And then can run any of the bitbake commands:

```
:~/belk/build$ bitbake memtester
```

The resulting packages (the default format is ipk) can be found inside `build/tmp/deploy/ipk`.

3.4.7 ConfigID

ConfigID is a new feature of DAVE Embedded Systems products. It's main purpose is providing an automatic mechanism for the identification of the product model and configuration.

With ConfigID, we aim at:

- completing the hardware configuration information that the software can't normally auto-detect (i.e. RAM chip version,...), implementing a dedicated reliable detect procedure
- when required, overriding the auto-detected hardware configuration information

When implemented, this mechanism allows for:

- initializing in the proper way the hardware platform, based on the specific features and parameters of the product, using a common software base (eg: a typical case is the SDRAM controller parameters, which must be configured by U-Boot depending on the particular memory chip, which can be different for the various SOM models)

- getting the complete hardware configuration (combining ConfigID with the information collectable at runtime) of a product deployed on the field

In simple words, model identification means the capability of reading a numerical code, stored in an available device (SOC's OTP , I2C EEPROM, 1-wire memories, protected NOR flash, etc.)

There are two **ConfigIDs**:

- SOM ConfigID: which reflects the characteristics of the SOM (stored on the SOM itself)
- Carrier Board (CB) ConfigID: which reflects the characteristics of the carrier board that hosts the SOM (stored on the carrier board itself and read by the SOM at boot time)

An additional attribute is **UniqueID**, which is a read-only code which univocally identifies a single product and is used for traceability.

3.4.7.1 Customer's action

DAVE Embedded Systems recommends to be up-to-date with Official SOM's BSPs for taking advantages of ConfigID/UniqueId features: this is the only required action.

- ConfigID advantage: to allow U-Boot bootloader to be executed only with the correct configuration (if the U-Boot loaded is not the proper one, it may stop execution avoiding incorrect behaviour)
- UniqueID advantage: to trace univocally each individual SOMs and, in turn, all the on-the-field equipments

3.4.7.2 ConfigID values

ConfigID is a N-bit (typically N>8) signed integer, that can have the following values:

- < 0: error
  - -1: not initialized
- = 0: ConfigID legacy
  - for prototypes (ConfigID not yet defined) or for products manufactured before the introduction of the ConfigID feature
- > 0: valid ConfigID
  - values are reported accordingly with the specific product table

3.4.7.3 ConfigID hardware implementation on BORA/BORAX

BORA/BORAX uses the first 32bytes OTP block on NOR SPI to store ConfigID (and its CRC32), UniqueID (and its CRC32)

3.4.7.4 ConfigID software implementation on BORA/BORAX

U-Boot integrates the software routines for reading and displaying the ConfigID. Hereunder an example of SOM ConfigID at startup:

```
U-Boot 2014.07 (Jul 24 2015 - 14:30:55) [belk-2.2.0]


Board:  BORA
I2C:   ready
DRAM:  ECC disabled 1 GiB
NAND:  1024 MiB
MMC:   zynq_sdhci: 0
SF: Detected S25FL256S_64K with page size 256 Bytes,
erase size 64 KiB, total 32 MiB
In:    serial
Out:   serial
Err:   serial
SOM ConfigID#: 00000002
SOM UniqueID#: fffffefc:fffffefc
```

```
CB ConfigID#: 00000001
CB UniqueID#: a600000f:24188b2d
Net:   Gem.e000b000
zynq-uboot>
```

For accesing these information on Linux procfs, the device tree must be modified (using u-boot fdt command): for example:

```
zynq-uboot> print loadfdt configid_fixupfdt
loadfdt=tftpboot ${fdtaddr} ${fdtfile}
configid_fixupfdt=if configid checkfdt ${fdtaddr}
som_configid ${som_configid#}; then if configid
checkfdt ${fdtaddr} cb_configid ${cb_configid#}; then
configid fdt_uniqueid ${fdtaddr}; fi; fi


zynq-uboot> run loadfdt configid_fixupfdt
Gem.e000b000 Waiting for PHY auto negotiation to
complete....... done
Using Gem.e000b000 device
TFTP from server 192.168.0.13; our IP address is
192.168.0.77
Filename 'bora/bora.dtb.as'.
Load address: 0x2000000
Loading: T ##
         1000 Bytes/s
done
Bytes transferred = 10019 (2723 hex)
FDT: property som_configid FDT: override
'som_configid' with '00000002'
FDT: property cb_configid match
FDT: override 'som_uniqueid' with 'fffffefc:fffffefc'
FDT: override 'cb_uniqueid' with 'a600000f:24188b2d'
```

It is possible to read the ConfigID/UniqueID via procfs; for example:

```
root@bora:~# for f in  /proc/device-tree/*id*; do echo
```

```
-n "$f: "; cat $f; echo; done
/proc/device-tree/cb_configid: 00000001
/proc/device-tree/cb_uniqueid: a600000f:24188b2d
/proc/device-tree/som_configid: 00000002
/proc/device-tree/som_uniqueid: fffffefc:fffffefc
df646299:0b0579d4
root@axel-lite:~#
```

3.4.8 System boot and recovery via microSD card

BELK provides a bootable microSD that can be used not only to quickly start the system, but also as a recovery method in case the primary boot device (eg. SPI NOR flash) gets erased or corrupted. The following sections describe how to create a bootable SD card and how to configure the system for booting from SD.

3.4.8.1 How to create a bootable MicroSD card

This section describes how to create a new bootable microSD card from scratch.

The following components must be available:

- FSBL built with Vivado 2014.4 as described in Section 3.4.2
- U-boot built in elf format, as described in Section 3.4.3
- FPGA bitstream (optional)

The procedure is the following:

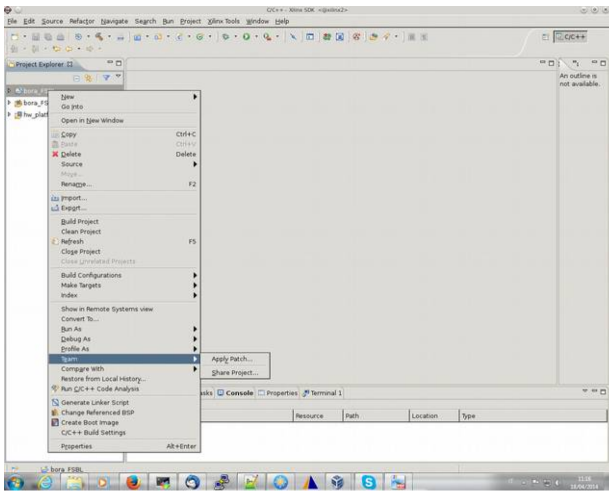
- from the Vivado 2014.4 SDK, apply the required patches to the `main.c` project file.This step can be done in two ways:
  - manually, directly modifying the `main.c` file adding the following code snippets:

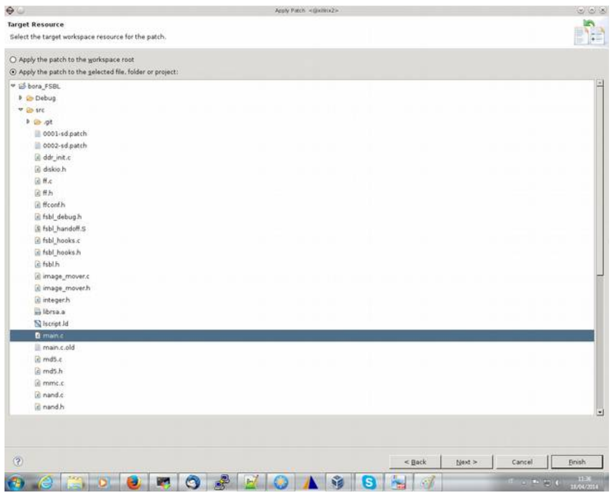
```
----CUT----
      /*
       * Unlock SLCR for SLCR register write
       */
      SlcrUnlock();
```

```
+       *((u32 *)0xF8000830) = 0x003F003F;
+       *((u32 *)0xF8000834) = 0x003F003F;

        /* If Performance measurement is required
         * then read the Global Timer value , Please note
that the
----CUT----
```

```
----CUT----
        /*
         * Read bootmode register
         */
        BootModeRegister = Xil_In32(BOOT_MODE_REG);
        BootModeRegister &= BOOT_MODES_MASK;

+       // always init QSPI
+       InitQspi();

        /*
         * QSPI BOOT MODE
         */
#ifdef XPAR_PS7_QSPI_LINEAR_0_S_AXI_BASEADDR
----CUT----
```

- automatically, using the **Apply Patch** function and selecting the `belk-sd-boot.patch` file provided with the BELK (please refer to the following images):





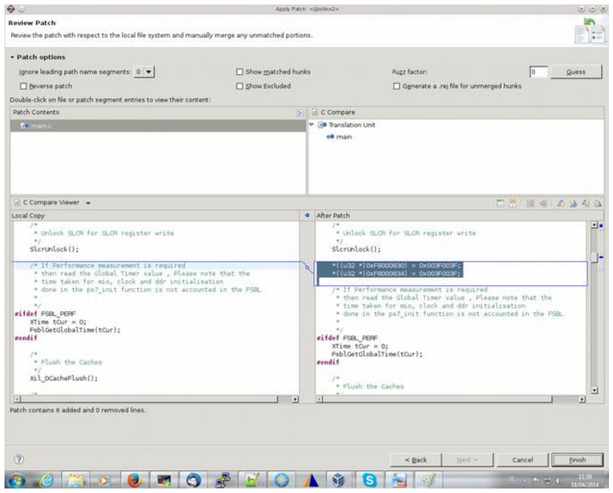
Apply Patch <@xilinx2>

Target Resource

Select the target workspace resource for the patch.

( ) Apply the patch to the workspace root
(•) Apply the patch to the selected file, folder or project:

```
▾ bora_FSBL
  ▸ Debug
  ▾ src
    ▸ .git
      0001-sd.patch
      0002-sd.patch
      ddr_init.c
      diskio.h
      ff.c
      ff.h
      ffconf.h
      fsbl_debug.h
      fsbl_handoff.S
      fsbl_hooks.c
      fsbl_hooks.h
      fsbl.h
      image_mover.c
      image_mover.h
      integer.h
      librsa.a
      lscript.ld
      main.c
      main.c.old
      md5.c
      md5.h
      mmc.c
      nand.c
      nand.h
```



Apply Patch <@xilinx2>

Review Patch

Review the patch with respect to the local file system and manually merge any unmatched portions.

▾ Patch options

Ignore leading path name segments: 0 ☐ Show matched hunks Fuzz factor: 0 Guess

☐ Reverse patch ☐ Show Excluded ☐ Generate a .rej file for unmerged hunks

Double-click on file or patch segment entries to view their content:

Patch Contents: main.c

C Compare: Translation Unit / main

C Compare Viewer

Local Copy:

```
    /*
     * Unlock SLCR for SLCR register write
     */
    SlcrUnlock();

    /* If Performance measurement is required
     * then read the Global Timer value , Please note that the
     * time taken for mio, clock and ddr initialisation
     * done in the ps7_init function is not accounted in the FSBL
     *
     */
#ifdef FSBL_PERF
    XTime tCur = 0;
    FsblGetGlobalTime(tCur);
#endif

    /*
     * Flush the Caches
     */
    Xil_DCacheFlush();
```

After Patch:

```
    /*
     * Unlock SLCR for SLCR register write
     */
    SlcrUnlock();

    *((u32 *)0xF8000830) = 0x003F003F;
    *((u32 *)0xF8000834) = 0x003F003F;

    /* If Performance measurement is required
     * then read the Global Timer value , Please note that the
     * time taken for mio, clock and ddr initialisation
     * done in the ps7_init function is not accounted in the FSBL
     *
     */
#ifdef FSBL_PERF
    XTime tCur = 0;
    FsblGetGlobalTime(tCur);
#endif

    /*
     * Flush the Caches
```

Patch contains 6 added and 0 removed lines.

- once the patch are applied, rebuild the FSBL project
- from the **Xilinx Tools** menù, select **Create Zynq Boot Image**
- select **Create New BIF file** and insert path and name of the .bif file
- in the **Boot image partitions** section, click on **Add** to browse and add the following files:
  - FSBL in .elf format, with partition type *"bootloader"*
  - (optional) FPGA bitstream in .bit format, with partition type *"datafile"*

    - U-boot binary with .elf extension, with partition type *"datafile"*
- in the **Output path** section, browse and select the path where saving the boot.bin file
- on a PC, format the microSD card creating a FAT32 partition
- copy the boot.bin file to the microSD card FAT32 partition

3.4.8.2 How to configure the system for SD boot

The S5 dip-switch on the BORAEVB/BORAXEVB carrier boards is used for configuring the boot mode. Please refer to the following table to select the MicroSD mode (highlighted in green):

	S5 dip-switch positions							
Boot mode	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
MicroSD	OFF	ON	OFF	ON	ON	OFF	ON	OFF
NOR SPI	OFF	ON	OFF	ON	ON	ON	ON	OFF
NAND	OFF	ON	OFF	ON	ON	OFF	ON	ON

4 Frequently Asked Questions

4.1 Q: Where can I found BORA/BORAX SOM information?

A: please refer to the following table:

Document	Location
BORA main page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/Category:Bora
BORAX main page on **DAVE Embedded Systems** Developers Wiki	http://wiki.dave.eu/index.php/Category:BoraX
BORA Hardware Manual	http://www.dave.eu/sites/default/files/files/bora-hm.pdf
BORAX Hardware description	http://wiki.dave.eu/index.php/Category:BoraX#Hardware
BORA product page	http://www.dave.eu/products/som/xilinx/zynq-XC7Z010-XC7Z020_bora
BORAX product page	http://www.dave.eu/products/som/xilinx/zynq-XC7Z015-XC7Z030_bora-xpress

4.2 Q: I've received the BELK package. How am I supposed to start working with it?

A: You can follow the steps listed below:

1. Check the kit contents with the packing list included in the box
2. Connect the power supply adapter and the serial cable
3. Start your terminal emulator program

4. Make sure the microSD card provided with the development kit is inserted into the BORAEVB/BORAXEVB slot
5. Switch on the power supply
6. Monitor the boot process on the serial console
7. Install and configure the development environment

4.3 Q: How can I update the BELK version?

A: please refer to section 3.3.3.5.

4.4 Q: Which software components am I supposed to install to start working with BORA/BORAX?

A: the following software packages must be installed on the Zynq development server:

- Vivado® Design Suite version **2014.4**
- Xilinx Software Development kit
- Python 2.7.x (C:\Python27 must be the installation directory on Windows)
- A Git tool (e.g. for Windows: MsysGit (http://msysgit.github.io/))

These tools can be installed either on Windows or Linux operating systems. However, if you need to build U-Boot or Linux, or use features like booting from NFS, the development server should be a Linux host. Please note that the required disk space for a full installation (BELK, Xilinx development tools, source trees, ...) is approximatively 25 GB.

4.5 Q: Can you suggest some guidelines for the carrier board design?

A: As a starting point, you can refer to the Wiki page dedicated to the carrier board design guidelines (http://wiki.dave.eu/index.php/Carrier_board_design_guidelines_%28SOM%29), that will highlight some best practices that applies to all SOMs. For specific information on BORA/BORAX, please refer to the BORA/BORAX Integration Guide (

http://wiki.dave.eu/index.php/Integration_guide_%28Bora/BoraX%29)

4.6 Q: Why my cross-compiled application doesn't work with the pre-packaged root file system provided with BELK?

A: as a general rule, dynamically linking an application against libraries built with a different toolchain can cause malfunctioning in the application. Since this pre-built root file system is not generated using the same cross-toolchain used for building the BELK software components, we recommend to choose one of the following options:

- if a native compiler is available on the root file system, go for native compilation instead of cross-compilation
- when you cross-compile, rely on static linking and avoid dynamic linking against the root file system libraries
- build your application using the same cross-toolchain (when available) used for building the root file system

4.7 Q: do you provide some application notes?

A: yes, we are actively publishing additional contents regarding applications for the BORA/BORAX platform. Please refer to the following page for the latest application notes: http://wiki.dave.eu/index.php/Application_Notes_%28Bora%29.

4.8 Q: Can I use BORA/BORAX in an Asymmetric Multi Processing (AMP) configuration?

A: Yes, Zynq processor can be configured to run independent software stacks on each of its processor cores. BELK owners can get access to an application note that describes how to build the software components required to run a simple application on FreeRTOS running on the second Zynq core, while Linux runs on the first Zynq core. For further details, please refer to http://wiki.dave.eu/index.php/Application_Notes_%28Bora%29

4.9 Q: I've just installed the tools on Linux and when I launch the "vivado" command, the build procesess fails. How can I solve this problem?

A: Please check the files bora.runs/bora_run_synth/runme.log, vivado.log, <vivado_build.log>. If you see messages like

```
INFO: [Common 17-78] Attempting to get a license: Synthesis
WARNING: [Common 17-301] Failed to get a license: Synthesis
....
....
Starting synthesis...

INFO: [Common 17-347] Attempting to get a license for feature
'Synthesis' and/or device 'xc7z020'
WARNING: [Common 17-348] Failed to get the license for
feature 'Synthesis' and/or device 'xc7z020'
1 Infos, 1 Warnings, 0 Critical Warnings and 0 Errors
encountered.
synth_design failed
ERROR: [Common 17-345] A valid license was not found for
feature 'Synthesis' and/or device 'xc7z020'. Please run the
Xilinx License Configuration Manager for assistance in
determining
which features and devices are licensed for your system.
...
```

```
...
...
Failed to create directory:
'/home/user/.Xilinx/Vivado/2013.3'
Failed to create the shortcut directory:
'/home/user/.Xilinx/Vivado/2013.3/shortcuts'
Failed to create the layout directory:
'/home/user/.Xilinx/Vivado/2013.3/layouts/application'
...
...
```

```
...
...
...
Run output will be captured here:
/home/user/bora/bora-build-20131115-nobk/bora.runs/bora_run_i
mpl/runme.log
[Fri Nov 15 11:41:07 2013] Waiting for bora_run_impl to
finish...
```

```
[Fri Nov 15 11:41:15 2013] bora_run_impl finished
wait_on_run: Time (s): cpu = 00:00:00.43 ; elapsed = 00:00:08
. Memory (MB): peak = 589.242 ; gain = 3.996
Traceback (most recent call last):
  File "fpga-bit-to-bin.py", line 25, in <module>
    bitfile = open(args.bitfile, 'rb')
IOError: [Errno 2] No such file or directory:
'/home/user/bora/bora-build-20131115-nobk/bora.runs/bora_run_
impl/bora_design_wrapper.bit'
    while executing
"exec $cmd fpga-bit-to-bin.py --flip
$proj_dir/bora.runs/bora_run_impl/bora_design_wrapper.bit
$proj_dir/bora.runs/bora_run_impl/bora_design_wrapper.bi..."
    invoked from within
"if {$bitstream == "-bitstream"} {
puts "Generating BITSTREAM"
reset_run -quiet bora_run_impl
reset_run -quiet bora_run_synth
launch_runs -verbose ..."
    (file "build_project.tcl" line 113)
Vivado%
```

please check the permissions of the `/home/user/.Xilinx` directory and make sure that the actual user has full access to that directory:

```
$ ll -d /home/user/.Xilinx/
drwxrwxrwx 6 root root 4096 Apr 17  2013 /home/user/.Xilinx//
```

4.10 Q: How can I configure the BORA/BORAX system to boot from network?

A: booting from network is very helpful during the software development (both for kernel and applications). The kernel and device tree binary images are downloaded via TFTP while the root file system is remotely mounted via NFS from the host. It is assumed that the development host:

- is connected with the target host board through an Ethernet LAN
- exports the directory containing the root file system for the target through the NFS server
- runs a TFTP server.
- has a proper subnet IP address

If your system does not match this configuration, just change the necessary variables and store them permanently with the u-boot `setenv`/`saveenv` commands. To do that, from the U-boot shell, please check the following parameters and set them accordingly with your host and target configuration:

Parameter	Description	Default
serverip	IP address of the host machine running the tftp/nfs server	192.168.0.23
ipaddress	IP address of the target	192.168.0.60
ethaddr	MAC address of the target	00:50:c2:1e:af:af
netmask	Netmask of the target	255.255.255.0
gatewayip	IP address of the gateway	192.168.0.254
netdev	Ethernet device name	eth0
rootpath	Path to the NFS-exported directory	/home/belk/rfs/bora
bootfile	Path to the kernel binary image on the tftp server	belk/uImage
fdtfile	Path to the device tree binary image on the tftp server	belk/bora.dtb
nfsargs	Kernel command line with parameters for loading the root file system through NFS	setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath} rootdelay=2

To run this configuration just enter the command

```
run net_nfs
```

5 Appendixes

5.1 U-Boot environment

The following is the u-boot environment that can be printed using the `print` command (please note this is an example and that the actual variables may differ for different U-Boot settings):

```
addcons=setenv bootargs ${bootargs} console=${console},115200n8 debug earlyprintk
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gateway}:${netmask}:$
{hostname}:${ethdev}
addmisc=setenv bootargs ${bootargs} maxcpus=${nr_cpus} cma=16M
baudrate=115200
bootaddr=0x101100
bootcmd=run ${modeboot}
bootdelay=3
bootenv=uEnv.txt
bootfile=bora/uImage
configid_fixupfdt=if configid checkfdt ${fdtaddr} som_configid ${som_configid#}; then if
configid checkfdt ${fdtaddr} cb_configid ${cb_configid#}; then configid fdt_uniqueid $
{fdtaddr}; fi; fi
console=ttyPS0
ethact=zynq_gem
ethaddr=00:50:C2:1E:AF:E0
ethdev=eth0
fdt_base=0x00780000
fdt_high=0x1F000000
fdtaddr=0x02000000
fdtfile=bora/bora.dtb
fpga_base=0x00180000
fpgafile=bora/bora.bin
fsbl_base=0x40000
gateway=192.168.0.254
header_base=0
hostname=bora
importbootenv=echo Importing environment from mmc ...; env import -t ${loadaddr} ${filesize}
ipaddr=192.168.0.89
jtag_vx=run vxargs; bootvx 0x200000
kernel_base=0x00800000
load=tftpboot ${loadaddr} bora/u-boot.bin
loadaddr=0x02080000
loadbootenv=fatload mmc ${mmcdev} ${loadaddr} ${bootenv}
loadfdt=tftpboot ${fdtaddr} ${fdtfile}
loadfpga=tftpboot ${loadaddr} ${fpgafile}
loadfsbl=tftpboot ${loadaddr} bora/bora_fsbl.bin
loadhdr=tftpboot ${loadaddr} bora/boot_header
loadk=tftpboot ${loadaddr} ${bootfile}
loadvx=tftpboot ${loadaddr} ${vxfile}
mmcargs=setenv bootargs root=/dev/mmcblk0p${mmcpart} rw
mmcdev=0
mmcpart=2
net_nfs=run loadk loadfdt nfsargs addip addcons addmisc; if run configid_fixupfdt; then
bootm ${loadaddr} - ${fdtaddr}; fi
net_vx=run loadvx vxargs; bootvx ${loadaddr}
netmask=255.255.255.0
nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath},tcp
nr_cpus=2
program_fpga=run loadfpga;fpga load 0 ${loadaddr} 0x${filesize}
qspiboot=echo Booting from QSPI: use net_nfs && run net_nfs
ramdisk_size=0x200000
rootpath=/opt/nfsroot/bora/belk
sdboot=if mmcinfo; then run uenvboot; echo Copying Linux from SD to RAM... && fatload mmc 0
```

```
${loadaddr} ${bootfile} && fatload mmc 0 ${fdtaddr} ${devicetree_image} && run mmcargs
addcons && run configid_fixupfdt && bootm ${loadaddr} - ${fdtaddr}
serverip=192.168.0.13
stderr=serial
stdin=serial
stdout=serial
u-boot_base=0x80000
uenvboot=if run loadbootenv; then echo Loaded environment from ${bootenv}; run
importbootenv; fi; if test -n $uenvcmd; then echo Running uenvcmd ...; run uenvcmd; fi
update=sf probe 0 0 0;sf erase ${u-boot_base} +${filesize};sf write ${loadaddr} $
{u-boot_base} ${filesize}
updatefdt=sf probe 0 0 0;sf erase ${fdt_base} +${filesize};sf write ${fdtaddr} ${fdt_base} $
{filesize}
updatefpga=sf probe 0 0 0;sf erase ${fpga_base} +${filesize};sf write ${loadaddr} $
{fpga_base} ${filesize}
updatefsbl=sf probe 0 0 0;sf erase ${fsbl_base} +${filesize};sf write ${loadaddr} $
{fsbl_base} ${filesize}
updatehdr=sf probe 0 0 0;sf erase ${header_base} +${filesize};sf write ${loadaddr} $
{header_base} ${filesize}
updatek=sf probe 0 0 0;sf erase ${kernel_base} +${filesize};sf write ${loadaddr} $
{kernel_base} ${filesize}
vxargs=setenv bootargs gem(0,0)host:vxWorks.st h=${serverip} e=${ipaddr} g=${gatewayip} tn=$
{hostname} u=target pw=vxTarget f=0x0
vxfile=bora/vxWorks

Environment size: 3301/262139 bytes
```