Bora Embedded Linux Kit (BELK)

AN-BELK-001

Asymmetric Multiprocessing (AMP) on Bora – Linux + FreeRTOS
# Table of Contents

1 Preface ......................................................................................... 5  
  1.1 About this document ................................................................. 5  
  1.2 Copyrights/Trademarks ............................................................... 5  
  1.3 Standards .................................................................................. 5  
  1.4 Disclaimers ............................................................................... 5  
  1.5 Technical Support .................................................................. 5  
  1.6 Related documents .................................................................. 7  
  1.7 Conventions, Abbreviations, Acronyms ...................................... 8  
2 Introduction .................................................................................. 10  
  2.1 Bora SOM .............................................................................. 10  
  2.2 Xilinx Zynq 7000 SOC .............................................................. 12  
  2.3 Asymmetric Multiprocessing ..................................................... 13  
  2.4 BELK .................................................................................... 13  
3 AMP on Bora .............................................................................. 15  
  3.1 Prerequisites ........................................................................... 15  
  3.2 Building the software components ........................................... 15  
  3.2.1 Vivado project ................................................................... 15  
  3.2.2 FSBL .................................................................................. 17  
  3.2.3 FreeRTOS applications ....................................................... 19  
      3.2.3.1 Importing the FreeRTOS repository into the SDK .......... 19  
      3.2.3.2 Building Example #1: HelloWorld application ........ 21  
      3.2.3.3 Building Example #2: RPMsg-based application ....... 24  
  3.2.4 Linux Device Tree .............................................................. 25  
3.3 Running the demo applications .................................................. 27  
  3.3.1 Example #1: HelloWorld application .................................... 27  
  3.3.2 Example #2: RPMsg-based application ............................... 28  
4 Advanced debugging techniques for AMP Linux+FreeRTOS configuration ................................................. 31  
  4.1 Introduction ............................................................................ 31  
  4.2 Prerequisites .......................................................................... 32  
  4.3 TRACE32 configuration ............................................................ 32  
  4.3.1 Multicore configuration ....................................................... 33  
  4.3.2 Multicore synchronization ................................................... 33  
  4.4 Startup scripts ........................................................................ 34  
  4.4.1 PRACTICE macros for multiple TRACE32 ......................... 34  
  4.4.2 The SYNch command ......................................................... 35  
  4.5 Setting up the Linux debug configuration ............................... 38  
      4.5.1 Kernel awareness ............................................................ 39  
      4.5.2 MMU support .................................................................. 40  
      4.5.3 Debugging of kernel modules ......................................... 41  
      4.5.4 Debugging of user processes, threads, shared objects .... 41  
  4.6 Setting up the FreeRTOS debug configuration ......................... 42
### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7 Onchip trace</td>
<td>43</td>
</tr>
<tr>
<td>4.8 Summary view</td>
<td>45</td>
</tr>
<tr>
<td>4.9 Lauterbach References</td>
<td>46</td>
</tr>
<tr>
<td>5 Appendixes</td>
<td>47</td>
</tr>
<tr>
<td>5.1 U-Boot environment</td>
<td>47</td>
</tr>
<tr>
<td>5.1.1 Example #1: HelloWorld application</td>
<td>47</td>
</tr>
<tr>
<td>5.1.2 Example #2: RPMMsg-based application</td>
<td>48</td>
</tr>
</tbody>
</table>
1 Preface

1.1 About this document
This application note describes how to run a simple application on FreeRTOS porting for Zynq, running on Zynq core #2.

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 document that are not strictly part of the Bora CPU module and the BoraEVB carrier board.
Bora 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 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.

### 1.6 Related documents

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>BELK Quick Start Guide</td>
<td>Provided with BELK</td>
</tr>
</tbody>
</table>

**Tab. 1:** Related documents
1.7 Conventions, Abbreviations, Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMP</td>
<td>Asymmetric multiprocessing</td>
</tr>
<tr>
<td>BELK</td>
<td>Bora Embedded Linux Kit</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GPI</td>
<td>General purpose input</td>
</tr>
<tr>
<td>GPIO</td>
<td>General purpose input and output</td>
</tr>
<tr>
<td>GPO</td>
<td>General purpose output</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PL</td>
<td>Zynq Programmable Logic</td>
</tr>
<tr>
<td>PS</td>
<td>Zynq Processing System</td>
</tr>
<tr>
<td>PSU</td>
<td>Power supply unit</td>
</tr>
<tr>
<td>SOC</td>
<td>System-on-chip</td>
</tr>
<tr>
<td>SOM</td>
<td>System-on-module</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric multiprocessing</td>
</tr>
</tbody>
</table>

Tab. 2: Abbreviations and acronyms used in this manual
# Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.0</td>
<td>November 2013</td>
<td>First release</td>
</tr>
<tr>
<td>1.0.1</td>
<td>November 2013</td>
<td>Added UART0 pinout information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor fixes</td>
</tr>
<tr>
<td>1.1.0</td>
<td>November 2013</td>
<td>Added support for RPMsg example</td>
</tr>
<tr>
<td>1.5.0</td>
<td>December 2013</td>
<td>Added chapter related to Lauterbach debugger</td>
</tr>
<tr>
<td>1.5.1</td>
<td>January 2014</td>
<td>Minor fixes</td>
</tr>
<tr>
<td>1.6.0</td>
<td>April 2014</td>
<td>Minor fixes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated for BELK 2.0.0 release</td>
</tr>
</tbody>
</table>
2 Introduction

2.1 Bora SOM

BORA is the new top-class Dual Cortex-A9 + FPGA CPU module by DAVE Embedded Systems, based on the recent Xilinx Zynq XC7Z010/XC7Z020 application processor.

Thanks to BORA, customers are going to save time and resources by using a compact solution that includes both the CPU and the 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.

BORA 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 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 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 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, please refer to Bora Hardware Manual.
2.2 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.
2.3 Asymmetric Multiprocessing

Asymmetric Multi Processing (AMP) allows a multiprocessor system to run multiple Operating Systems (OS) that are independent of each other. In other words, each CPU has its own private memory space, which contains the OS and the applications that are to run on that CPU. In addition, there can be some shared memory space that is used for multiprocessor communication. This is contrasted with Symmetric Multiprocessing (SMP), in which one OS runs on multiple CPUs using a public shared memory space. Thanks to AMP, developers can use open-source Linux and FreeRTOS operating systems and the RPMMsg Inter Processor Communication (IPC) framework between the Zynq's two high-performance ARM® Cortex™-A9 processors to quickly implement applications that need to deliver deterministic, real-time responsiveness for markets such as automotive, industrial and others with similar requirements. For further information, please refer to http://www.wiki.xilinx.com/Multi-OS+Support+%28AMP+%26+Hypervisor%29

2.4 BELK

Bora 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-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.

Please refer to the **BELK Quick Start Guide** for further details on BELK.

**N.B.**: this application note has been tested using BELK 2.0.0.
3 AMP on Bora

The following sections describe how to build the software components required to set up asymmetric multi-processing (AMP for short) configuration required to run Linux OS on first Cortex-A9 core and FreeRTOS on second Cortex-A9 core.

Two different examples are provided. The first one – HelloWorld – shows basic functionalities while the second – RPMsg-based application – exploits more sophisticated techniques to handle inter-processors communication and synchronization. This latter configuration is based on RPMsg mechanism as described in Xilinx document UG978 (v2013.04, April 22, 2013).

3.1 Prerequisites

- Vivado® Design Suite version 2013.3 with Xilinx SDK (Webpack license is minimum requirements)
- Python 2.7.x (C:\Python27 must be the installation directory on Windows)
- Bora Embedded Linux Kit (Please refer to BELK Quick Start Guide for further details) version 2.0.0.
- BORA FreeRTOS repository (please refer to section 3.2.3)

3.2 Building the software components

3.2.1 Vivado project

- 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 (the “-b” option is used to automatically checkout the current branch):
git clone
git@git.dave.eu:dave/bora/bora.git -b bora

● Enter the git directory
● Switch to bora branch (not required if this is already the current branch): git checkout bora

● Set project directory variable:
  export PROJ_DIR=$(pwd)/../bora-build-YYYYMMDD-nobk

● Configure Vivado settings:
  . /opt/Xilinx/Vivado/2013.3/settings64.sh

● Launch Vivado with build_project script:
  vivado -mode tcl -source build_project.tcl
  -notrace -tclargs "-bitstream"

---
1 In a 32 bit system, Vivado settings are configured with the following command
   /opt/Xilinx/Vivado/2013.3/settings32.sh
2 Passing the -tclargs "-bitstream" parameters allows for automatic building of the FPGA bitstream.
3.2.2  FSBL

Once the Vivado project build is completed, the hardware configuration can be exported starting the SDK to build the FSBL. From the SDK GUI:

- Create a new application project, as shown in the picture below:

- Configure the application settings as shown in the pictures below:
● Click finish to launch FSBL build process
● Create the binary from the FSBL ELF choosing one of the following options:
  ■ 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
  ■ 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**

**N.B.** When the Vivado project is modified, the binary must be re-generated with the following command:

```python
python 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.bin
```

### 3.2.3 FreeRTOS applications

The following sections describe the steps required to configure and build both the Helloworld and the RPMsg-based examples.

#### 3.2.3.1 Importing the FreeRTOS repository into the SDK

● Assuming that a local repository has not been created, clone the remote FreeRTOS git repository:

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

● Enter the git directory

● Switch to **freertos-AMP** branch: `git checkout freertos-AMP`
In SDK gui import new repository: Xilinx Tools -> Repositories

Click New... to add a new repository under Local or Global Repositories, and select the freeRTOS repository directory:
Click Rescan Repositories, Apply and OK

At the end of the procedure, applications based on freeRTOS operating system can be built

### 3.2.3.2 Building Example #1: HelloWorld application

The first example shows basic AMP functionalities. On FreeRTOS side, UART0 is used to implement a simple console. This port is routed via EMIO signals to pin-strip connector of BoraEVB. Since these signals are driven by FPGA Bank #34, these pins are 3.3V. Thus a RS232 transceiver or an USB/UART bridge should be used in order to connect the console on a PC. The signals are routed to the JP17 connector of the BoraEVB as reported below:

- **JP17.4** – UART0_TX
- **JP17.6** – UART0_RX

Please follow the steps listed below to build a HelloWorld application that prints a message on UART0 (via EMIO) on May, 2014.
FreeRTOS running on Bora core #2.

- From the SDK GUI, create a new application project:

- Configure the application settings as shown in the pictures and table below:
### Project Details

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project name</td>
<td>helloworld_freeRTOS</td>
</tr>
<tr>
<td>Hardware Platform</td>
<td>hw_platform_0</td>
</tr>
<tr>
<td>Processor</td>
<td>ps7_cortexa9_1</td>
</tr>
<tr>
<td>OS Platform</td>
<td>freertos_zynq</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
</tr>
<tr>
<td>Board Support Package</td>
<td>Create New</td>
</tr>
<tr>
<td>Type</td>
<td>FreeRTOS Hello World AMP template</td>
</tr>
</tbody>
</table>

- Click finish to launch the application build process
- Create the binary from the application ELF choosing one of the following options:
  - manually launch the command:
    ```
    arm-xilinx-eabi-objcopy -v -O binary 
    $PROJ_DIR/bora.sdk/SDK/SDK_Export/helloworld_freeRTOS/Debug/helloworld_freeRTOS.elf
    $PROJ_DIR/bora.sdk/SDK/SDK_Export/helloworld_freeRTOS.elf
    ```
configure the automatic binary generation on project build. In Project Explorer, right-click on “helloworld_freeRTOS” 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

3.2.3.3 Building Example #2: RPMsg-based application

The procedure needed to build this application is similar to the one used to build HelloWorld application. The only difference is that the FreeRTOS Latency AMP template must be selected.

In this case please note that:

● the standard Linux infrastructure will be used to load the firmware for the second core
● Linux will start in SMP mode, running on both cores; then CPU1 will be shutdown and freeRTOS firmware will be loaded and run

This application exploits TTC1 timer to measure IRQ latencies as described in Xilinx UG978. In addition to that, GPIO0 (pin JP21.16 on BoraEVB) will be toggled every time ISR is invoked.

Once the build process is completed, the executable file in .elf format will be generated (we suggest to name it freertos). Creating the .bin file is not required.

<table>
<thead>
<tr>
<th>Project name</th>
<th>RPMsg_freeRTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Platform</td>
<td>hw_platform_0</td>
</tr>
<tr>
<td>Processor</td>
<td>ps7_cortexa9_1</td>
</tr>
<tr>
<td>OS Platform</td>
<td>freertos_zynq</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
</tr>
<tr>
<td>Board Support</td>
<td>Create New</td>
</tr>
<tr>
<td>Type</td>
<td>FreeRTOS Latency AMP</td>
</tr>
</tbody>
</table>
To run this example, Linux kernel\(^3\) must be rebuilt too\(^4\). First of all copy the freertos executable file in .elf format (freertos) into the directory firmware of Linux kernel tree\(^5\). Then configure the kernel using bora_amp_defconfig as configuration file and enter the following command line, that changes the default load address of kernel and launches the building of both the kernel image and the modules:

```bash
bash# make UIMAGE_LOADADDR=0x10008000 uImage modules
[...]
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
UIMAGE arch/arm/boot/uImage
Image Name: Linux-3.9.0-bora-1.1.0-xilinx-00
Created: Thu Nov 21 15:55:07 2013
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 3217192 Bytes = 3141.79 kB = 3.07 MB
Load Address: 10008000
Entry Point: 10008000
Image arch/arm/boot/uImage is ready
```

The file arch/arm/boot/uImage is the binary image of the kernel that must be used to boot the system.

The following kernel modules, resulting from the kernel build procedure, must be copied from the building directory to the root file system (usually into /lib/modules/<kernel version>/kernel, but any other directory can be used):

```plaintext
LD [M]  drivers/remoteproc/remoteproc.ko
LD [M]  drivers/remoteproc/zynq_remoteproc.ko
LD [M]  drivers/rpmsg/rpmsg_freertos_statistic.ko
LD [M]  drivers/rpmsg/virtio_rpmsg_bus.ko
LD [M]  drivers/virtio/virtio.ko
LD [M]  drivers/virtio/virtio_ring.ko
LD [M]  net/rpmsg/rpmsg_proto.ko
```

For further details on kernel modules, please refer to [http://tldp.org/HOWTO/Module-HOWTO/](http://tldp.org/HOWTO/Module-HOWTO/)

### 3.2.4 Linux Device Tree

The Flattened Device Tree (FDT) is a data structure for describing the hardware in a system (for further information, please refer to [http://elinux.org/Device_Tree](http://elinux.org/Device_Tree)). Both Example #1

---

\(^3\) The kernel branch must be bora.

\(^4\) It is assumed that the development environment is already set up as described in BELK Quick Start Guide.

\(^5\) The name of the binary file copied into the firmware directory must be freertos
and Example #2 requires some modifications to the standard Bora device tree (to initialize UART0 port and to properly initialize the RPMsg infrastructure, respectively). Please use the kernel branch bora, that already includes the aforementioned patches (for further details, please refer to the arch/arm/boot/dts/bora.dts file and commit descriptions on the Linux git repository). For detailed instructions on how to build the Linux kernel and the Device Tree, please refer to the BELK Quick Start Guide (Section 3.4.3).
3.3 Running the demo applications

3.3.1 Example #1: HelloWorld application

This section describes how to run freeRTOS HelloWorld example application on BORA using AMP (Linux + FreeRTOS). Please follow the steps listed below:

- Place all the binary files into the host tftp directory:

  | Kernel6 | uImage |
  | Device Tree | bora.dtb |
  | First stage bootloader | bora_FSBL.bin |
  | FPGA bitstream | bora_design_wrapper.bin |
  | FreeRTOS application | helloworld_freeRTOS.bin |

- Start the Bora system

- From the U-Boot shell, update the FSBL with the following commands:
  
  ```
  run load_fsbl
  run update_fsbl
  ```

- Reset the board to reboot with the new FSBL

- Add the following U-Boot environment variables:

  ```
  setenv addcons 'setenv bootargs ${bootargs} console=${console},115200n8 cma=16M debug maxcpus=${nr_cpus}'
  setenv addmem 'setenv bootargs ${bootargs} mem=${kernel_mem}'
  setenv kernel_mem 1008M
  setenv nr_cpus 1
  setenv net_nfs 'run program_fpga; run load_freertos;
  run loadk nfsargs addip addcons addmem; bootm ${loadaddr_kern} - ${loadaddr_ftd}';
  setenv load_freertos 'tftp ${freertos_addr} ${freertos_file};mw.l 0xFFFFFFF0 ${freertos_addr}';
  ```

6 The kernel must be built with the UIMAGE_LOADADDR 0x8000 option. Please refer to section 3.4.3 of the Belk Quick Start Guide

7 program_fpga: Loads FPGA binary from TFTP and programs the bitstream
load_freertos: Loads freertos application binary from TFTP and writes application start address for core#2
mem=${kernel_memory}: sets maximum kernel memory (1008M = 1024M - 16M)
maxcpus=${nr_cpus}: sets maximum Linux cpus to 1
setenv freertos_addr 0x3F000000
setenv freertos_file bora/BELK/helloworld_freeRTOS.bin
setenv fpga_file BELK/bora_design_wrapper.bin

- Boot the system running the following command:
  run net_nfs

3.3.2 Example #2: RPMsg-based application

As stated before, this example shows a more sophisticated approach that allows for:

- using a standardized communication channel between the two cores
- exploiting a standardized mechanism to load the firmware of second core

The example performs IRQ latency measurements on FreeRTOS side by using a hardware timer. These measures are collected by the counterpart application running on Linux side and shown on console. Please follow the steps listed below:

- Place all the binary files into the host tftp directory:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>uImage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Tree</td>
<td>bora.dtb</td>
</tr>
<tr>
<td>First stage bootloader</td>
<td>bora_FSBL.bin</td>
</tr>
<tr>
<td>FPGA bitstream</td>
<td>bora_design_wrapper.bin</td>
</tr>
<tr>
<td>FreeRTOS application</td>
<td>freertos</td>
</tr>
</tbody>
</table>

- Start the Bora system
- From the U-Boot shell, update the FSBL with the following commands:
  run load_fsbl
  run update_fsbl
- Reset the board to reboot with the new FSBL
- Add the following U-Boot environment variables:

8 program_fpga: Loads FPGA binary from TFTP and programs the bitstream
load_freertos: Loads freertos application binary from TFTP and writes application start address for core#2

9 Please note that, using the RPMsg mechanism, it's not required to set the maxcpus=${nr_cpu} variable
setenv addcons 'setenv bootargs \${bootargs} console=${console},115200n8 cma=16M debug'
setenv addmem 'setenv bootargs \${bootargs} mem=${kernel_mem}'
setenv kernel_mem 496M
setenv net_nfs 'run program_fpga; run loadk nfsargs addip addcons addmem; bootm \${loadaddr_kern} - \${loadaddr_ftd}'
setenv freertos_addr 0x3F000000
setenv fpga_file BELK/bora_design_wrapper.bin

● Boot the system running the following command:
run net_nfs

When booting, the Linux kernel will print out the following message to indicate it has been relocated to address 0x10000000:

```
[    0.000000] Machine: Xilinx Zynq Platform, model: Bora
[    0.000000] Change memory bank to 10000000-2fffffff
[    0.000000] cma: CMA: reserved 16 MiB at 2f000000
```

To start the example, please enter the following commands on Linux side to load the required modules:

```
ismod drivers/virtio/virtio.ko
ismod drivers/virtio/virtio_ring.ko
ismod drivers/rpmsg/virtio_rpmsg_bus.ko
ismod net/rpmsg/rpmsg_proto.ko
ismod drivers/remoteproc/remoteproc.ko
ismod drivers/remoteproc/zynq_remoteproc.ko
ismod drivers/rpmsg/rpmsg_freertos_statistic.ko
```

Linux kernel will print these messages, informing that the communication between the two cores has been established:

```
[   17.966158] NET: Registered protocol family 41
[   18.036698] CPU1: shutdown
[   18.045287] remoteproc0: 0.remoteproc-test is available
[   18.050522] remoteproc0: Note: remoteproc is still under development and considered experimental.
[   18.059554] remoteproc0: THE BINARY FORMAT IS NOT YET FINALIZED, and backward compatibility isn't yet guaranteed.
[   18.077341] remoteproc0: powering up 0.remoteproc-test
[   18.082668] remoteproc0: Booting fw image freertos, size 2357682
[   18.103607] remoteproc0: remote processor 0.remoteproc-test is now up
[   18.113339] virtio_rpmsg_bus virtio0: rpmsg host is
Then run the `latencystat` application as shown below. The typical output will look like this:

```bash
root@bora:~# ./latencystat -b
Linux FreeRTOS AMP Demo.
  0: Command 0 ACKed
  1: Command 1 ACKed
Waiting for samples...
  2: Command 2 ACKed
  3: Command 3 ACKed
  4: Command 4 ACKed

----
Histogram Bucket Values:
  Bucket 323 ns (36 ticks) had 38 frequency
  Bucket 341 ns (38 ticks) had 299 frequency
  Bucket 512 ns (57 ticks) had 1 frequency
  Bucket 746 ns (83 ticks) had 1 frequency

----
Histogram Data:
  min: 323 ns (36 ticks)
  avg: 332 ns (37 ticks)
  max: 746 ns (83 ticks)
  out of range: 0
  total samples: 339

----
```

This application is extremely useful for evaluating how CPU load on first core affects IRQ latency. In case latency does not satisfy real-time requirements, it may be necessary to adjust arbitration priorities of processor's interconnect subsystem. For further details, please refer to chapter “Interconnect” of Zynq Reference Manual.

**N.B.** prior to launching the `latencystat` application, make sure that the governor is set to "performance" with the following command:

```bash
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```
4 Advanced debugging techniques for AMP Linux+FreeRTOS configuration

4.1 Introduction

When working with complex real-time configurations such as AMP Linux+FreeRTOS, debugging requirements increase dramatically. This chapter – written in collaboration with Lauterbach SRL (www.lauterbach.it) – shows how these issues can be tackled with Lauterbach TRACE32® debugger.

The following picture shows the BoraEVB connected to Lauterbach PowerDebug Interface/USB3 via J18 connector. By default, the board is configured to chain Xilinx PL TAP and ARM DAP (please refer to chapter “JTAG and DAP Subsystem”.

Fig. 3: BoraEVB connected to Lauterbach PowerDebug Interface/USB3

10 The techniques described in this chapter apply to the “Example #1: HelloWorld” FreeRTOS application (please refer to section 3.2.3.2).

The following sections describe in detail how to configure TRACE32 ® debugger to support debug of Linux running on the first Zynq core, and FreeRTOS, running on the second Zynq core.

4.2 Prerequisites

- LA-3500 Power Debug USB3 or LA-7705 Power Debug Ethernet or LA-7699 PowerDebug II
- LA-7843 JTAG Debugger for Cortex-A/-R
- LA-7960X License for Multicore Debugging
- **Optional**: LA-7970X Trace License for ARM (Debug Cable)

For a general introduction to debug features provided by TRACE32 tools, please refer to:

- “Debugger Basics – Training” manual (training_debugger.pdf)
- “Training HLL Debugging” manual (training_hll.pdf)

4.3 TRACE32 configuration

In AMP configuration, each core runs a unique code, already fixed at compile time. The CPU interoperates with other processing units, exchanging data through dedicated channels (for example, shared memory buffers or peripheral units). Lauterbach supports these architectures with different TRACE32 instances, each one connected to a single core, in “core view” configuration where debug focus is on single processor.

However, as the cores do not work independently but perform the application task together and in parallel, it is possible to start and stop all the cores simultaneously. This is the only way to test the interaction between the cores and to monitor and control the entire application. Moreover, as each core run a
separate part of the application, the majority of the symbol and debug information is assigned exclusively to the corresponding core.

In the following paragraphs, the basic TRACE32 multicore configuration for a single device will be introduced. For more details, please refer to “ICD Debugger User's Guide” (debugger_user.pdf).

### 4.3.1 Multicore configuration

For the configuration of TRACE32 application, reference scripts are provided from Lauterbach. The first GUI must be started manually and must register itself to share a common JTAG handler with other TRACE32 applications. This is done setting the option \texttt{CORE=} in the configuration file (default file name: config.t32).

```plaintext
PBI= ; within config file of first core
USB
CORE=1

or:

PBI= ; within config file of first core
NET
NODE=<IP_address>
PACKLEN=1024
CORE=1
```

Nevertheless, the setting to define which core is addressed, actually is done later on.

### 4.3.2 Multicore synchronization

To use the start/stop synchronization between different core debuggers, the INTERCOM port settings are necessary. This is done assigning predefined port numbers in the configuration file to each TRACE32 application (option \texttt{PORT=}).

```plaintext
IC=NETASSIST ; within config file of first core
first core
PORT=20001
```

May, 2014
4.4 **Startup scripts**

In order to use a generic configuration file for each TRACE32 instance, there is the possibility to use just one generic template file for all cores. The particular settings are passed as parameter. This is shown in the reference script:

```bash
amp_start_core0.bat
```

which refers to the configuration file:

```bash
amp_config.t32
```

The batch file starts automatically this startup script:

```bash
amp_demo.cmm
```

After booting the first TRACE32 GUI, the second GUI will be started automatically by the startup script. See the reference script:

```bash
amp_demo_start_core1.cmm
```

For more details about PRACTICE batch language, please refer to:

- “Training PRACTICE” manual (training_practice.pdf)
- “PRACTICE Script Language User’s Guide” (practice_user.pdf)
- “PRACTICE Script Language Reference Guide” (practice_ref.pdf)

4.4.1 **PRACTICE macros for multiple TRACE32**

The startup script, started automatically at the first TRACE32 application, is fully able to configure the whole debug system, providing PRACTICE commands both to the current instance of TRACE32 application, and to the second instance. It’s also possible to deliver the same PRACTICE command to both instances with a single command line. The command redirection is possible using the INTERCOM feature. Typically
some PRACTICE macros can be defined for this purpose.

```plaintext
&core0="" ;only to improve readability
&core1="intercom localhost:&intercomport_core1"
&both="GOSUB intercom_both"
```

where:

```plaintext
intercom_both:
    LOCAL &param
    ENTRY %Line &param
    &core0 &param
    &core1 &param
    RETURN
```

In this way, all CPU-specific configuration commands can be performed in the same way for each TRACE32 application, or distinguishing between different configurations. For example:

```plaintext
&both SYStem.RESet
```

or:

```plaintext
&core0 SYStem.CPU ZYNQ-7000CORE0
&core1 SYStem.CPU ZYNQ-7000CORE1
```

### 4.4.2 The SYnch command

The synchronization between different TRACE32 applications is done by SYnch command group, which allows the following purposes:

- to establish a start/stop synchronization between the cores controlled by different TRACE32 instances;
- to allow concurrent assembler single steps between the cores controlled by different TRACE32 instances;
- to allow synchronous system mode changes between the cores controlled by different TRACE32 instances.
The SYnch settings are reported below:

**Connect**
Establish a connection to the debugger attached to the defined communication port(s). Several debuggers ports can be specified, separated by space.

**MasterGo ON**
If the program execution is started, the program execution for all other processors which have SlaveGo ON is also started.

**MasterBrk ON**
If the program execution is stopped, the program execution for all other debuggers which have SlaveBrk ON is also stopped.

**MasterStep ON**
If an asm single step is executed, all processors which have SlaveStep ON will also asm single step.

**MasterSystemMode ON**
Invite other TRACE32 instances to perform system mode changes synchronously. System mode changes are typically performed by the commands SYStem.Mode <mode>

**SlaveGo ON**
The program execution is started, if a processor with MasterGo ON starts its program execution.
SlaveBrk ON
The program execution is stopped, if a processor with MasterBrk ON stops its program execution.

SlaveStep ON
A asm single step is performed, if a processor with MasterStep On performs an asm single step.

SlaveSystemMode ON
Synchronize with system mode changes in connected TRACE32 instances.

A summary of SYnch configuration is provided with command TargetSystem, which also allows to easily and rapidly modify the SYnch mode options.

Moreover the TargetSystem command provides a general overview of the whole multicore configuration and of the current state.

```
TargetSystem DEFault Title SYnch.All
InterComPort /Global
```

There is a time delay between reaction of different cores. The
reaction time of the slave core depends on the technical realisation of the synchronization. If no specific configuration is performed, the synchronization is done by software (e.g., the master debugger informs the slave debugger via socket communication on the host about specified events). On the other side, if the on-chip Cross Trigger Interface is configured, the synchronization takes place directly on the processor. This is a faster solution and the time delay between reaction of different cores becomes around 0-10ns.

The CTI interface for chip Zynq is configured with the following commands. If CTI is configured, TRACE32 will use the faster solution for the synchronization.

```
&core0 SYStem.CONFIG.CTI.Base 0x80098000
&core0 SYStem.CONFIG.CTI.Config CORTEXV1

&core1 SYStem.CONFIG.CTI.Base 0x80099000
&core1 SYStem.CONFIG.CTI.Config CORTEXV1
```

### 4.5 Setting up the Linux debug configuration

The symbolic information is useful for HLL debugging, or setting breakpoints, stepping through the code, viewing variables, and many other aspects of debugging. The compiler must be configured in order to generate debug symbols. The `vmlinux` file for the running kernel must be available, in order to load the kernel debug symbols.

No instrumentation is needed in the kernel source code for debugging with Lauterbach, but it’s important that the `vmlinux` file is generated from the same kernel build as the `zImage` or `uImage` running on the system.

The `Data.LOAD` command is used to load the kernel symbols, and the `sYmbol.SourcePATH` command can be used, if necessary, to define additional search directories for the source files. The `Data.LOAD` command for `vmlinux` is applied to `&core0`, so that symbol information is not shared with the other `TRACE32` application.
Specific options must be configured to avoid automatic Break of TRACE32 debugger, in case any of the following events happens due to normal Linux operations.

- \texttt{TrOnchip.Set UNDEF OFF} ; may be used by Linux for FPU detection
- \texttt{TrOnchip.Set DABORT OFF} ; used by Linux for page miss!
- \texttt{TrOnchip.Set PABORT OFF} ; used by Linux for page miss!

In the following paragraphs, the basic TRACE32 Linux configuration will be introduced. For more details, please refer to:

- “Training Linux Debugging” manual (training_rtos_linux.pdf)

### 4.5.1 Kernel awareness

TRACE32 kernel awareness technology makes debugger aware of the OS running in the target system. Debug is significantly
simplified, as the user can immediately access all the components of the OS and the application. The Executable and Linkable Format (ELF) binaries, created at kernel build time, are used also by Linux awareness.

The Linux kernel awareness is configured with the commands:

```bash
; loads Linux awareness
&core0 TASK.CONFIG
~/demo/arm/kernel/linux/linux-3.x/linux3.t32
; loads Linux menu
&core0 MENU.ReProgram
~/demo/arm/kernel/linux/linux-3.x/linux.men
```

The Linux kernel awareness is loaded into the first TRACE32 application, and will not affect the second one.

The Linux menu file (`linux.men`) includes many useful menu items developed for the TRACE32 GUI to ease Linux debugging.

### 4.5.2 MMU support

In Linux embedded, the Lauterbach debuggers provide a very tight integration with the RTOS. The kernel awareness supports Linux MMU format and is able to handle virtual memory addressing.

```bash
MMU.FORMAT LINUX swapper_pg_dir
0xc0000000--0xc1fffffff 0x00000000
TRANSlation.Create 0xc0000000--0xc1fffffff
0x00000000
TRANSlation.COMMON 0xbf000000--0xffffffff
TRANSlation.TableWalk ON
TRANSlation.ON
```

where the virtual address range is the virtual-to-physical kernel address mapping, according with results of command `MMU.List KernelPageTable`, executed when the kernel is up and running.
In the command `TRANSlation.COMMON`, the virtual address range has been extended below the kernel start address, because kernel objects are loaded in this memory range.

### 4.5.3 Debugging of kernel modules

The Linux kernel can be compiled to allow linking of additional modules at runtime (kernel objects). The Lauterbach debuggers also support kernel modules debugging, starting from the initialization function.

### 4.5.4 Debugging of user processes, threads, shared objects

User process debugging is also available, starting from the very beginning of the process. If the process loads shared objects, they are loaded in the process address space when the related instructions are executed for the first time (demand paging).

The Lauterbach debuggers also support debug of threads for multithreaded processes. In this case, the same address space is shared between different threads and the symbolic information can be loaded only once per process.

In general, the same techniques used for debugging kernel code, such as setting breakpoints, stepping through code, watching variables, and viewing memory contents, can be performed in the same way for processes and tasks.

Virtual address spaces are distinguished in TRACE32 using the concept of spaceID, which is enabled with the option:

```
SYStem.Option MMUspaces ON
```

The memory addressing is extended using the lower 16 bit of
the process PID, allowing in this way to distinguish between equal virtual addresses for different processes.

4.6 Setting up the FreeRTOS debug configuration

A similar kernel awareness concept as Linux, is provided by Lauterbach for many other RTOS. Among the others, also FreeRTOS is supported with the following awareness configuration.

```plaintext
; load FreeRTOS awareness
&core1 TASK.CONFIG
~/demo/arm/kernel/freertos/freertos.t32
```
The FreeRTOS kernel awareness is loaded into the second TRACE32 application, and will not affect the first one.

As of symbol information it’s enough to load the proper symbol file, redirecting the Data.LOAD command to &core1. In this way, the second TRACE32 application will have visibility of FreeRTOS HLL debug info.

The FreeRTOS menu file (freertos.men) includes useful menu items developed for the TRACE32 GUI to ease FreeRTOS debugging.

For more details, please refer to “RTOS debugger for FreeRTOS” manual (rtos_freertos.pdf).

4.7 Onchip trace

The Zynq chip implements the ETB (Embedded Trace Buffer), a CoreSight hardware component providing on-chip trace functionality. The ETB stores program-flow trace information on-chip at high rates and at 32-bit data width. The data can be read out via JTAG, when the trace recording has ended.

This trace method is enabled in TRACE32 with the following commands, respectively performed on each GUIs of AMP configuration:

```
; load FreeRTOS menu
&core1 MENU.ReProgram
~/demo/arm/kernel/freertos/freertos.men
```

```
Trace.METHOD Onchip ; select the ETB as source
                     ; for the trace information
```
The TRACE32 menu item “Trace ? Configuration” allows a full control of configuration, initialization and listing of trace information.

The trace, recorded for each core by each AMP TRACE32 GUI, can be synchronized using the XTRACK feature of TRACE32. In this way, an easy comparison is possible of program flow of each core at the same time.

The XTRACK feature is enabled with the following commands:

```
&core0 Synch.XTRACK
localhost:&intercomport_core1
```
4.8 Summary view

In the pictures below, all the concepts previously discussed are shown as a summary global view of TRACE32 debugger, respectively for Zynq core 0 running a Linux kernel, and Zynq core 1 running a FreeRTOS based application.

Fig. 4: Linux at Zynq core 0
4.9 Lauterbach References

This chapter has been written by Lauterbach italian branch office.

Contact information:
Lauterbach SRL
Via Enzo Ferrieri 12
20153 Milan (Italy)
Tel. +39 02 45490282
Email info_it@lauterbach.it
Web www.lauterbach.it
5 Appendixes

5.1 U-Boot environment

The following is the U-Boot environment that can be printed using the `print` command (please note these are examples and that the actual variables may differ for different U-Boot settings), for both the FreeRTOS applications described in this application note.

5.1.1 Example #1: HelloWorld application

```
addcons=setenv bootargs ${bootargs} console=${console},115200n8 cma=16M debug maxcpus=${nr_cpus}
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gateway}:${netmask}:${hostname}:${ethdev}
addmem=setenv bootargs ${bootargs} mem=1008M baudrate=115200 bootdelay=3
bootfile=bora/uImage
console=ttyPS0 dt_base=0x005C0000 ethact=Gem.e000b000 ethaddr=00:0a:35:00:01:22
ethdev=eth0 fdt_high=0x1F000000 fileaddr=0 filesize=2f0ed0 fpga_base=0x00180000
fpga_file=belk/free_rtos/bora_design_wrapper.bin freertos_addr=0xF0000000 freertos_file=bora/belk/free_rtos/helloworld_freertos.bin
fsbl_base=0x400000 ftd_file=bora/bora.dtb gateway=192.168.0.254 header_base=0
hostname=bora ipaddr=192.168.0.209 jtagboot=echo TFTPing Linux to RAM...;tftp 0x8000 zImage;tftp 0x10000000 devicetree.dtb;tftp 0x800000 ramdisk8M.image.gz;go 0x8000 kernel_size=0x140000 load=tftp ${loadaddr} bora/u-boot.bin
load_dt=tftp ${loadaddr} bora/bora.dtb load_fpga=tftp ${loadaddr} bora/${fpga_file} load_freertos=tftp ${freertos_addr} ${freertos_file};mw.l 0xFFFFFFF0 ${freertos_addr} load_fsbl=tftp ${loadaddr} bora/belk/free_rtos/bora_FSBL.bin load_header=tftp ${loadaddr} bora/boot_header-1.1.0 loadaddr=0x08000000 loadaddr_ftd=0x01000000
loadaddr_kern=0x0
load_k=tftp ${loadaddr_kern} ${bootfile};tftp ${loadaddr_fdt} ${fdt_file}
modeboot=qspiboot net_nfs=run program_fpga;run load_freertos;run loadk nfsargs addip addcons addmem;bootm ${loadaddr_kern} - ${loadaddr_fdt} netmask=255.255.255.0 nfsargs=setenv bootargs root=/dev/nfs rw nfsroot=${serverip}:${rootpath}
qspiboot=sf probe 0 0 0;sf read 0x8000 0x100000 0x2c0000;sf read 0x1000000 0x3c0000
0x40000;sf read 0x8000000 0x4000000 0x8000000;go 0x8000
```
5.1.2 Example #2: RPMsg-based application

```
addcons=setenv bootargs ${bootargs} console=${console},115200n8 cma=16M debug
addip=setenv bootargs ${bootargs} ip=${ipaddr}:${serverip}:${gateway}:${netmask}:httpd:hostname:${ethdev}
addmem=setenv bootargs ${bootargs} mem=496M
baudrate=115200
bootdelay=3
bootfile=bora/uImage
console=ttysPS0
dt_base=0x005C0000
ehtub=0x0000
ethaddr=00:0a:35:00:01:22
ethdev=eth0
dtb_high=0x1F000000
fileaddr=0
filesize=2f0ed0
fpga_base=0x00180000
fpga_file=belk/free_rtos/bora_design_wrapper.bin
freertos_addr=0x3F000000
freertos_file=bora/belk/free_rtos/helloworld_freertos.bin
fsbl_base=0x40000
ftd_file=bora/bora.dtb
gateway=192.168.0.254
header_base=0
hostname=bora
ipaddr=192.168.0.209
jtagboot=echo TFTPing Linux to RAM...;tftp 0x8000 zImage;tftp 0x1000000 devicetree.dtb;tftp 0x800000 ramdiskK7M.image.gz;go 0x8000
kernel_size=0x140000
load=tftp ${loadaddr} bora/u-boot.bin
load_dt=tftp ${loadaddr} bora/bora.dtb
load_fpga=tftp ${loadaddr} bora/${fpga_file}
load_freertos=tftp ${freertos_addr} bora/${freertos_file}
load_fsbl=tftp ${loadaddr} bora/belk/free_rtos/bora_FSBL.bin
load_h=tftp ${loadaddr} bora/belk/free_rtos/bora_HDR.bin
loadaddr=0x08000000
loadaddr_ftd=0x01000000
loadaddr_kern=0x0
loadk=tftp {loadaddr_kern} ${bootargs};tftp {loadaddr_ftd} {ftd_file}
modeboot=qspiboot
net_nfs=run program_fpga; run loadk nfsargs addip addcons addmem; bootm {loadaddr_kern} - $ {loadaddr_ftd}
nr_cpus=1
program_fpga=run load_fpga;fpga load 0 $ {loadaddr} 0x$ {filesize}
qspiboot=sf probe 0 0 0;sf read 0x8000 0x100000 0x2c0000;sf read 0x1000000 0x3c0000 0x400000;sf read 0x800000 0x400000 0x800000;go 0x8000
ramdisk_size=0x200000
```
rootpath=/home/shared/devel/dave/bora-DBRx/sw/linux/rfs/BELK/ubuntu_12.04
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
serverip=192.168.0.23
stderr=serial
stdin=serial
stdout=serial
u-boot_base=0x80000
update=sf probe 0 0 0;sf erase ${u-boot_base} 0x80000;sf write ${loadaddr} ${u-boot_base} 0x80000
update_dt=sf probe 0 0 0;sf erase ${dt_base} 0x40000;sf write ${loadaddr} ${dt_base} 0x40000
update_fpga=sf probe 0 0 0;sf erase ${fpga_base} 0x440000;sf write ${loadaddr} ${fpga_base} 0x440000
update_fsbl=sf probe 0 0 0;sf erase ${fsbl_base} 0x40000;sf write ${loadaddr} ${fsbl_base} 0x40000
update_fsbl_nand=run load_fsbl; nand erase 0x40000 0x40000; nand write ${loadaddr} 0x40000 0x40000
update_h=sf probe 0 0 0;sf erase ${header_base} 0x40000;sf write ${loadaddr} ${header_base} 0x40000