Software Development Starter Guide

XM-ARM

Fent Innovative Software Solution

July, 2017
Reference: 14-035-03.009.sum.03

This page is intentionally left blank.

DOCUMENT CONTROL PAGE

TITLE: Software Development Starter Guide: XM-ARM

AUTHOR/S: Fent Innovative Software Solution

LAST PAGE NUMBER: 33

VERSION OF SOURCE CODE: XM-ARMv2.0.X for processor ARM CORTEX-A9()

REFERENCE ID: 14-035-03.009.sum.03

SUMMARY: This is a quick start guide that covers the main topics required to develop applications on XM-ARM for CORTEX-A9 processors.

DISCLAIMER:

REFERENCING THIS DOCUMENT:

```
@techreport {14-035-03.009.sum.03,
       title = {Software Development Starter Guide:  XM-ARM},
       author = { Fent Innovative Software Solution},
       institution = {Fent Innovative Software Solutions, S.L.},
       number = {14-035-03.009.sum.03},
       year={July, 2017},
}
```


Changes:

Version	Date	Author	Comments
01	09/06/2015	Javier Coronel	Initial Software Development Starter guide for XM-ARMv2.0
02	19/01/2016	Manuel Muñoz and Javier Coronel	Update Board configuration Update XM Building Requirements for XM-ARMv2.0
03	03/07/2017	Salva Peiro and Javier Coronel	Fix minor typos for XM-ARMv2.0.

Support

Javier Coronel (jcoronel@fentiss.com)
Miguel Masmano (mmasmano@fentiss.com)
FENT INNOVATIVE SOFTWARE SOLUTIONS, S.L.

Ciudad Politécnica de la Innovación
Camino de Vera s/n
CP: 46022
Valencia, Spain

The official XtratuM web site is: http://www.fentiss.com

Contents

List of Tables

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

Chapter 1

Introduction

In this chapter you will find information about the target audience, the objectives, the organization of this document, the related documents and some definitions and acronyms.

1.1 Audience

The target audience for this document is software developers that develop partitioned applications running on top of the XtratuM Hypervisor. You are expected to have some knowledge of ARINC-653 and experience in programming real-time applications.

1.2 Objectives

The purpose of this document is to serve as starter guide covering the main topics required to develop applications on XtratuM. Hence this document is a companion document of the XtratuM software development kit (XM-SDK for short) that guides through the XtratuM partition development process.

Covering the complete XtratuM partition development is out of the scope of this document, instead this document focus on the partition development using the XtratuM Abstraction Layer (XAL), which is a basic execution runtime for programming bare C based application.

1.3 Executive summary

This document is organized as follows:

- Chapter 1 presents the target audience, the objectives, the summary of this document, the related documents and some definitions and acronyms.
- Chapter 2 describes the required tools and the setup of the XtratuM development environment.
- Chapter 3 covers the programming of XAL partitions (XtratuM Abstraction Layer).
- Chapter 4 covers the execution of the of the final system image.

1.4 Related documents

Being a starter guide, this document is not aimed to cover all the aspects of XtratuM partition development, instead for more detailed documentation you are referred to the XtratuM user [2] and reference manuals [1] listed in the Bibliography section.

1.5 Terms, definitions and acronyms

1.5.1 Terms

communication port An inter-partition communication end point.

error An error is the part of the system state that may cause a subsequent failure. A failure occurs when an error reaches the service interface and alters the service.

hypercall The service (system call) provided by the hypervisor. The services provided are known as para-virtual services.

hypervisor The layer of software that, using the native hardware resources, provides one or more virtual machines (partitions).

partition Also known as "virtual machine" or "domain". It refers to the environment created by the hypervisor to execute user code.

partition code Also known as "guest". It is the code executed inside a partition. Usually, the code is composed of an operating system and a set of processes or threads. Since application code relies on the services provided by the OS, we will assume that the partition code is an operating system (or a real-time operating system).

resident software The booting software that is executed directly in ROM memory right after a system reboot, also referred to as boot-loader or firmware. Among other tasks, it is in charge of loading XtratuM and the initial partitions in RAM memory.

slot Amount of time that a partition can execute without other partition's preemption

1.5.2 Acronyms

Term	Description
API	Application Programming Interface.
bps	Bits Per Second.
CC	Common Criteria for Information Technology Security Evaluation.
ELF	Executable and Linkable Format.
RSW	Resident SoftWare.
RTEMS	Real-Time Executive for Multiprocessor Systems.
SD	Slot Duration.
SFP	Security Function Policy.
UART	Universal Asynchronous Receiver Transmitter. A serial port.
XAL	XtratuM Abstraction Layer.
XEF	XtratuM Executable Format.
XML	eXtended Markup Languaje.

Chapter 2

XtratuM Environment Setup

This chapter describes the required tools, performs the setup of the XtratuM development environment, that is the configuration, compilation and installation of XtratuM. In order to accomplish the previous goals, this chapter is organized as follows:

- Section 2.1 covers XtratuM development requirement tools.
- Section 2.2 covers the configuration of the XtratuM hypervisor.
- Section 2.3 covers the compilation of the XtratuM hypervisor.
- Section 2.4 covers the installation of the XtratuM hypervisor by means the Software Development Kit provided.

2.1 XtratuM requirements

The XtratuM development has been compiled and tested on a Debian stable Linux distribution (Kubuntu/Ubuntu 14.04 32/64 bits). Thus, from this point on, it is assumed that you will use the same operative system and Linux distribution. The following development tools have been used:

The *Type* column lists:

- packages marked as "*req*" which are **required** to compile XtratuM,
- while packages marked as "*opt*" are **optionally** needed to compile or use XtratuM.

The cross-compiler toolchain is provided for your comfort. This toolchain has been obtained from the Xilinx SDK 2014.3. You could use your own toolchain, but it is encouraged to use the provided toolchain since it is already tested.

2.2 XtratuM configuration

After ensuring that all the tools listed in the requirements tools section 2.1 are installed, the next step is to configure XtratuM, which is separated in three components configured as follows:

- Subsection 2.2.1 configures the XtratuM toolchain.

Package	Version	Linux package name	Type	Purpose
host gcc	4.6.8/4.8.2	gcc-4.6/gcc-4.8	req	Build host utilities
host binutils	2.24	binutils	req	Build host utilities
make	3.81-8	make	req	Core Building
makeself	2.2.0	makeself	req	Build self extracting distro
libncurses	5.9.20140118	libncurses5-dev	req	Configure source code
libxml2	2.7.8/2.9.1	libxml2-dev	req	Configure XtratuM parser
arm-xilinx-eabi-gcc	4.8.3	GCC(ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-ld	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-objcopy	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-as	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-ar	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
xmllint	2.7.8/2.9.1	libxml2-utils	req	Configure XtratuM parser
python	2.7.6	python	opt	Tools
perl	5.18.2	perl	opt	Testing
Xilinx SDK	2014.2/2014.3	Tools Xilinx SDK	req	Manage board connection, ARM Xilinx toolchain
uBoot	2011.03-dirty	uBoot provided by Xilinx	req	Board bootloader

Table 2.1: Summary of development tools required by XtratuM.

- Subsection 2.2.2 configures the XtratuM core.
- Subsection 2.2.3 configures the Resident Software bootloader.
- Subsection 2.2.4 configures the XAL basic partition execution environment.
- Subsection 2.2.5 explain how to use the default configuration for XtratuM and XAL.

2.2.1 The XtratuM toolchain configuration

The `xmconfig` file defines the target and host compilers for the target hardware architecture. A `xmconfig.arm` file is provided as template for the architecture (ARCH=arm). In order to configure the toolchain, simply copy the file as follows:

```
# configure (copy) the ARM architecture toolchain xmconfig
$ cp xmconfig.arm xmconfig
```

At this point would be advisable to check the value of some toolchain variables in the `xmconfig` file: `TARGET_CCPATH`: This variable must contain the path of the toolchain to be used. As example the variable looks like that:

```
TARGET_CCPATH=(PATH_TO_THE_TOOLCHAIN)/bin/
```

Note that if you have added the tool-chain path to the system path, you must maintain this variable empty.

`TARGET_CCPREFIX`: This variable must contain the prefix of the toolchain to be used. As example the variable looks like that:

```
TARGET_CCPREFIX=arm-xilinx-eabi-
```

After configuring the XtratuM toolchain, the next step is to configure XtratuM and XAL, which is performed by running the `make menuconfig` configuration utility. The following two subsections explain all the existing options of this utility to configure the XtratuM core and XAL basic partition execution environment in two different steps.

In order to use the default configuration without interactive menus, please refer to Subsection 2.2.5.

2.2.2 The XtratuM core configuration

After running `make menuconfig`, the utility will ask the user to press `Enter` to begin the configuration of XtratuM core.

```
$ make menuconfig

> Building Kconfig:

You have to configure three different elements:
1.- The XtratuM itself.
2.- The Resident Software, which is charge of loading the system from ROM
    -> RAM.
3.- The XAL, a basic partition execution environment.

Press 'Enter' to configure Xtratum (step 1)
```

Once `Enter` key is pressed, the XtratuM hypervisor core main menu of the utility will be shown. The configuration defines the target (processor and board), the physical memory layout, debug options, the basic hypervisor services and the virtualized drivers, among others.

Figure 2.1 shows a screen shot of the first step of menuconfig. The main menu options when configuring the XtratuM core are summarized in table 2.2. Notice in the table that the term XML configuration file has nothing to do with the configuration of XtratuM at this point. Once XtratuM is installed, the XML configuration file configures some parameters of the partitions, defines the system resources and establishes how they are allocated to each partition.

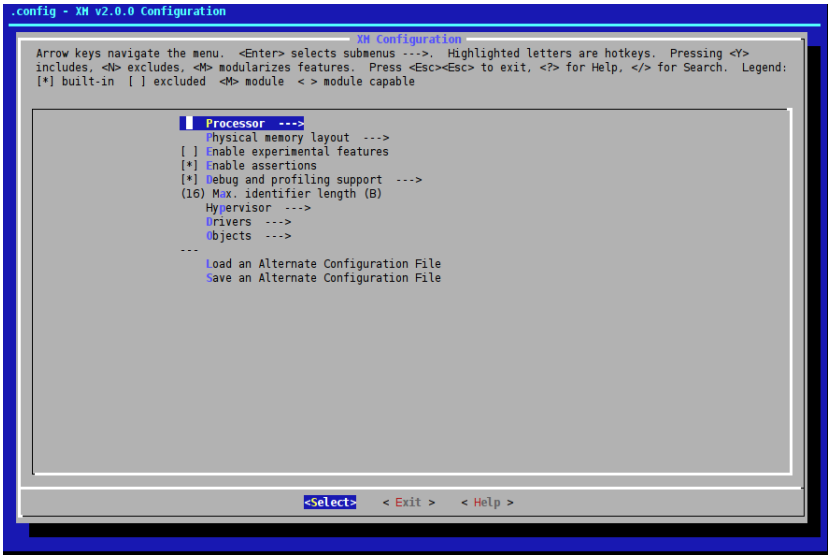

Figure 2.1: XtratuM core configuration.

Option	Purpose
Processor	Selects the target platform (processor, board and memory protection schema)
Physical memory layout	Sets the physical address where the hypervisor is located
Enable experimental features	Set of non-tested features
Enable assertions	Includes asserts in the XtratuM code
Debug and profiling support	Options when XtratuM is in debug mode
Max. identifier length	Indicates the maximum length that the identifiers of the XML configuration file must have
Hypervisor	Configures some parameters of the hypervisor
Drivers	Selects the devices managed by the hypervisor
Objects	Configures parameters related to console print and accounting

Table 2.2: Summary of XtratuM core configuration.

2.2.3 The Resident Software configuration

The Resident Software (Resident SW or RSW for short) is a small boot loader in charge of loading the XtratuM hypervisor and the partitions. Once `Enter` key is pressed, the RSW main menu of the utility will be shown. The main options when configuring the Resident Software are summarized in table 2.3:

Option	Purpose
Stack Size	Set the Resident Software Stack size
RSW memory layout / Load Address	Set the physical address where the Resident SW is loaded
RSW memory layout / Container Addr	Set the physical address where the container is located
Enable RSW output	The RSW will print diagnostic output to Serial port

Table 2.3: Summary of Resident Software configuration

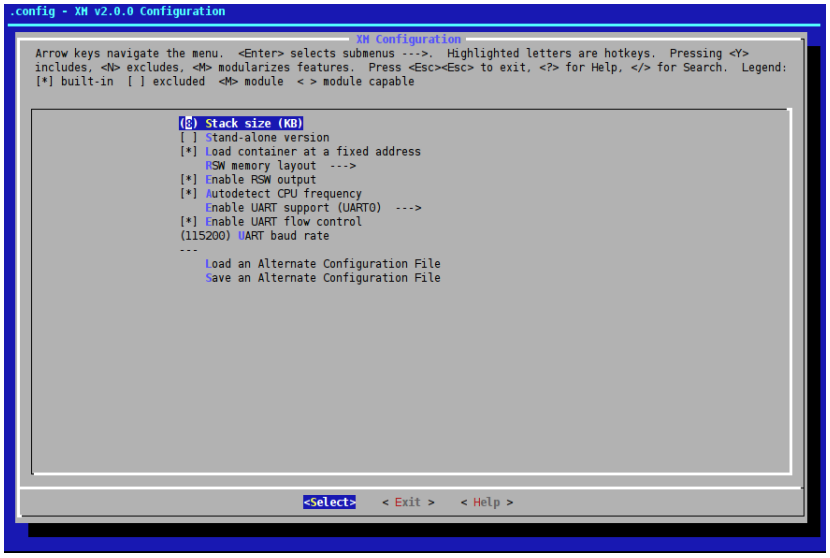

Figure 2.2: The Resident Software

2.2.4 The XAL configuration

After saving the XtratuM core and resident software configurations, step 3 is prompted.

```
$ make menuconfig

> Building Kconfig:

You have to configure three different elements:
1.- The XtratuM itself.
2.- The Resident Software, which is charge of loading the system from ROM
    -> RAM.
3.- The XAL, a basic partition execution environment.

Press 'Enter' to configure Xtratum (step 1)

> Target architecture: [arm]
Press 'Enter' to configure Resident Sw (step 2)
Press 'Enter' to configure XAL (step 3)
```

Once `Enter` key is pressed, the XAL main menu of the utility will be shown. The XAL is a bare runtime targeted to the development of basic C programming language partitions. The XAL configuration allows to specify a debug mode and the stack size of the partition.

Figure 2.3 shows a screen shot of the third first step of menuconfig. The main menu options when configuring the XAL are summarized in table 2.4.

Option	Purpose
Debug	Sets the XAL in debug mode
Kernel options	Stack size, which sets the XAL partitions user stack size Stack size for exceptions, which sets stack size used by the rutines of exceptions and interrupts

Table 2.4: Summary of XAL configuration.

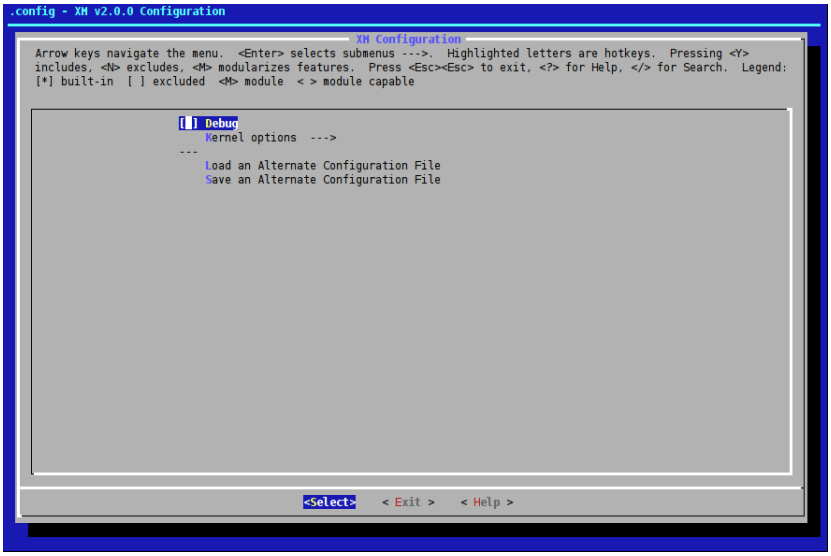

Figure 2.3: XAL configuration.

2.2.5 Using the configurations by default

The user can use a configuration defined by default for the ZEDBOARD board. The default configuration can be used through the command `make defconfig`.

```
$ make defconfig

> Building Kconfig:

> Target architecture: [arm]
```

Listing 2.1: Using configurations by default

This option of the `Makefile` configures XtratuM and XAL to be used with the ZEDBOARD board. Memory map of the XML files in the XAL examples is prepared for this default configuration.

2.3 XtratuM compilation

Once XtratuM has been configured, it can be built using the `make` utility.

```
$ make

> Configuring and building the "XtratuM hypervisor"
```

```
> Building XM Core
  - kernel/arm
  - kernel/mmu
  - kernel
  - klibc
  - klibc/arm
  - objects
  - drivers
> Linking XM Core
   text    data     bss     dec     hex filename
  54526     136   13972   68634   10c1a xm_core
0ddc79f76d7e53471c06d1896ca67782 xm_core.xef
> chmod XM Core
> Done

> Configuring and building the "User utilities"

> Building XM user
  - libxm
  - bootloaders/rsw
  - tools
  - tools/xmpack
  - tools/xmcparser
  - tools/xmbuildinfo
  - tools/rswbuild
  - tools/xef
  - xal
> Done
```

2.4 XtratuM Software Development Kit

The XtratuM Software Development Kit (XM-SDK for short) is a self-contained distribution containing the components required for XtratuM based development. It is packaged as an installer (`.run` file) to allow its installation by the partition developers.

The contents of the XM-SDK distribution are depicted in table 2.5:

Directory	Description
xm	XtratuM Core Virtualization
xal	XtratuM Abstraction Layer (XAL) for partition development
xal-examples	XtratuM examples with partitioned projects ready to use

Table 2.5: Contents of XM-SDK distribution.

- `xm-sdk-arm/xm` contains the XtratuM hypervisor core composed by the hypervisor kernel, the LibXM library providing the XtratuM services and resident software bootloader.
- `xm-sdk-arm/xal` contains the XtratuM Abstraction Layer (XAL) composed by the user tools, headers and XAL libraries for developing bare C partitions.

- `xm-sdk-arm/xal-examples` contains the XAL examples that are distributed together with their source code. The examples make use of the XAL to build the partitioned system.

2.4.1 XtratuM Software Development Kit build

The XM-SDK is built using the `make distro-run` command, which in addition performs the XtratuM compilation, so the `make` command for the XtratuM compilation can be skipped as shown in listing 2.4.1. A file with the pattern `xtratum-2.x.y.run`[1] as name is created after running successfully the command.

```
# make distro-run: performs the XtratuM compilation and creates the XM-SDK
    distribution
$ make distro-run

> Configuring and building the "XtratuM hypervisor"

> Building XM Core
  - kernel/arm
  - kernel/mmu
  - kernel
  - klibc
  - klibc/arm
  - objects
  - drivers
> Linking XM Core
   text    data     bss     dec     hex filename
 107554     256   65532  173342   2a51e xm_core
a7223ef65c451f2de7efd922d7fc2779 xm_core.xef
> XM Core assign attributes
> Done

> Configuring and building the "User utilities"

> Building XM user
  - libxm
  - bootloaders/rsw
  - tools
  - tools/xmpack
  - tools/xmcparser
  - tools/xmbuildinfo
  - tools/rswbuild
  - tools/xef
  - xal
> Done

> Installing XM in "/tmp/xtratum-2.0.0-17401/xtratum-2.0.0/xm"
   - Generating XM sha1sums
   - Installing XAL
   - Generating XAL sha1sums
   - Setting read-only (og-w) permission.
   - Deleting empty files/directories.
> Done
```

[1] x and y are numbers that mean the current subversion and revision of XtratuM.

```
> Generating XM distribution "xtratum-2.0.0.tar.bz2"
> Done

> Generating self extracting binary distribution "xtratum-2.0.0.run"
> Done
```

2.4.2 XtratuM Software Development Kit installation

As explained, the self-installer file `xtratum-2.x.y.run`[2] is installed running the command shown in 2.4.2. A proper installation location is under a directory with read-only permissions to prevent modifications. In this case `/home/<user>/xm-sdk-arm` has been selected.

```
$ ./xtratum-2.0.0.run -- -d /home/user/xm-sdk-arm/
Verifying archive integrity... All good.
Uncompressing XtratuM binary distribution 2.0.0: ......................
Starting installation.
Installation log in: /tmp/xtratum-installer-7260.log

1. Select the directory where XtratuM will be installed. The installation
   directory shall not exist.

2. Select the target compiler toolchain binary directory (arch arm).

3. Confirm the installation settings.

Important: you need write permision in the path of the installation
    directory.

Continue with the installation [Y/n]? Y

Press [Enter] for the default value or enter a new one.
Press [TAB] to complete directory names.

1.- Installation directory [/home/user/xm-sdk-arm/]:
2.- Path to the arch toolchain [/home/user/compilers/armel-cortexa9-eabi/
    bin/]:

Confirm the Installation settings:
Selected installation path : /home/user/xm-sdk-arm/
Selected toolchain path :  /home/user/compilers/armel-none-eabi/bin/

3.- Perform the installation using the above settings [Y/n]? Y

Installation completed.
```

As you may notice, passing the options `- -d /opt/xm-sdk-arm/` to `xtratum-2.x.y.run` makes this path as default path for the installation directory. Default paths are shown between square brackets.

After the installation, the `/opt/xm-sdk-arm/` folder is populated with the contents of the XM-SDK distribution.

[2] `x` and `y` are numbers that mean the current subversion and revision of XtratuM.

2.4.3 XtratuM Software Development Kit specific tools

Table 2.6 summarizes the tools distributed with the XM-SDK:

Tool	Location	Component	Purpose
xmbuildinfo	xm/bin/xmbuildinfo	Core	Bash script to get information from ELF file of XtratuM
xmcboottab	xm/bin/xmcboottab	Core	Creates the booting table for XtratuM. It informs the hypervisor about the partition loading information
xmcparser	xm/bin/xmcparser	Core	Builds system configuration from XML configuration file
xmeformat	xm/bin/xmeformat	Core	Handles the XtratuM XEF files
xminstall	xm/bin/xminstall	Core	Bash script to generate the XM-SDK installer
xmpack	xm/bin/xmpack	Core	Packs the XtratuM XEF files in a container
rswbuild	xm/bin/rswbuild	Resident SW	Build a bootable resident software
xpath	xal/bin/xpath	XAL	Parses the XML configuration file
xpathstart	xal/bin/xpathstart	XAL	Extracts memory address from XML configuration file

Table 2.6: XM-SDK distribution tools.

Chapter 3

XAL partition development

This chapter covers the XAL partition development. The development is presented starting with the *Hello World* XAL partition example. The rest of the chapter is organized as follows:

- Section 3.1 gives an overview of the XAL runtime.
- Section 3.2 covers the *Hello World* example distributed in the XM-SDK.
- Section 3.3 covers the programming of the *Hello World* XAL partition example.
- Section 3.4 summarizes the XAL partition development process.

3.1 XAL runtime overview

The XAL (XtratuM Abstraction Layer) is a minimal C runtime environment, that provides the following services:

- Partition initialization: Processor Modes, Trap Table, Stack, `PartitionMain` entry point.
- Minimal C library: `stdio`, `stdlib`, `string`.
- Additional libraries for IRQ management:
  - Individually set IRQs handlers (InstallIRQHandler).
  - IRQ enabling/disabling (HwSti, HwCli).
- Internally uses the LibXM hypercall API.

3.2 XAL *Hello World* example

The *Hello World* example is located at `/opt/xm-sdk-arm/xal-examples` along with other examples distributed with the XM-SDK. The contents before the compilation of the *Hello World* example directory are shown:

```
/home/user/xm-sdk-arm/xal-examples/hello_world
|-- Makefile          # A Makefile that automates the build process of the
    example.
```

```
|-- partition.c       # One or more source code files for the partitions
    code.
\-- xm_cf.arm.xml     # The XtratuM XML configuration file defining the
    partitioned system.
```

The `make` utility is used to perform the build of the example.

3.2.1 The Makefile

Following you will find an explanation about the contents of the Makefile (shown in listing 3.1) in order to see how the final image is constructed and to identify the main components.

- **XAL _PATH**: Defines the path of the XM-SDK used for development.
- **XMLCF**: Defines the XtratuM XML configuration file used by the example, shown in listing 3.2.2.
- **PARTITIONS**: Defines the partition or partitions code that composes the example, shown in listing 3.2.3.
- The rest of the Makefile defines how the system is built, using the Makefile rules provided by the XM-SDK.

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=../..

CONTAINER = container.bin
RSW = resident_sw

all: $(CONTAINER) $(RSW)

include $(XAL_PATH)/common/rules.mk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SRCS := $(sort $(wildcard *.c))
OBJS := $(patsubst %.c,%.o, $(SRCS)) $(patsubst %.S,%.o, $(ASRCS))

PARTITIONS=partition0.xef partition1.xef


partition0: $(OBJS) $(XMLCF)
	$(TARGET_LD) -o $@ $< $(TARGET_LDFLAGS) -Ttext=$(shell $(XPATHSTART
	    ) 0 $(XMLCF))

partition1: $(OBJS) $(XMLCF)
	$(TARGET_LD) -o $@ $< $(TARGET_LDFLAGS) -Ttext=$(shell $(XPATHSTART
	    ) 1 $(XMLCF))

PACK_ARGS=-h $(XMCORE):xm_cf.xef.xmc \
	-p 0:partition0.xef \
	-p 1:partition1.xef
```

```
$(CONTAINER): $(PARTITIONS) xm_cf.xef.xmc
	$(XMPACK) check xm_cf.xef.xmc $(PACK_ARGS)
	$(XMPACK) build $(PACK_ARGS) $@
```

Listing 3.1: XAL example Makefile.

Note the parameter `-Ttext` passed to the linker (`$(TARGET_LD)`) in each partition rule. They indicate to the linker which are the addresses of the beginning of the partition code section and data area.

As you can see in the XtratuM XML configuration file (listed in 3.2.2), two memory areas are defined, one for each partition. Depending on the parameters passed to `xpathstart` and `xpathmemarea`, a different memory area will be selected.

For `xpathstart` tool, the first parameter indicates the partition identifier, and it will select the first memory area defined for the selected partition. The second parameter is the path to the XtratuM XML configuration file.

However, despite `xpathmemarea` tool first and second parameter are the same than the ones of `xpathstart`, it has a third parameter which is used to select the memory area of the partition.

Thus, in the example, the first memory area defined for each partition in the XtratuM XML configuration file is used for the code section and the second is used for the data section. You can change the order of the memory areas to suit your needs. In this case you will need to change the parameters of `xpathstart` and `xpathmemarea` tools. Anyway it is advisable to choose a convention and use the same always.

3.2.2 The XtratuM XML configuration file

The XML configuration file defines the system resources and how they are allocated to each partition. Next listing shows the contents of the XtratuM XML configuration file.

```
<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version="
    1.0.0" name="hello_world">
    <HwDescription>
        <MemoryLayout>
            <Region type="rom" start="0x0" size="1MB" />
            <Region type="sdram" start="0x00100000" size="1023MB" />
        </MemoryLayout>
        <ProcessorTable>
            <Processor id="0" frequency="400Mhz">
                <CyclicPlanTable>
                    <Plan id="0" majorFrame="2000ms">
                        <Slot id="0" start="0ms" duration="250ms"
                            partitionId="0" />
                        <Slot id="1" start="500ms" duration="250ms"
                            partitionId="1" />
                        <Slot id="2" start="1s" duration="250ms" partitionId
                            ="0" />
                        <Slot id="3" start="1500ms" duration="250ms"
                            partitionId="1" />
                    </Plan>
                </CyclicPlanTable>
            </Processor>
        </ProcessorTable>
        <Devices>
```

```
            <Uart id="1" baudRate="115200" name="Uart" />
        </Devices>
    </HwDescription>
    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB" />
    </XMHypervisor>
    <PartitionTable>
        <Partition id="0" name="Partition0" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x10000000" size="256KB" />
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="Partition1" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x14000000" size="256KB" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>
</SystemDescription>
```

3.2.3 The partition source code

In order to program XAL partitions, the partition code has to include the following `#include` headers to use the services provided by the LibXM and LibXAL libraries.

- `#include <string.h>`
- `#include <stdio.h>`
- `#include <xm.h>`
- `#include <irqs.h>`

Next listing shows the partition code for the *Hello World* example:

```
/*
 * $FILE: partition.c
 *
 * Fent Innovative Software Solutions
 *
 * $LICENSE:
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
```

```
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307,
     USA.
 */

#include <string.h>
#include <stdio.h>
#include <xm.h>
#include <irqs.h>

#define PRINT(...) do { \
       printf("[P%d] ", XM_PARTITION_SELF); \
       printf(__VA_ARGS__); \
} while (0)

void PartitionMain(void)
{
    PRINT("Hello World!\n");
    XM_halt_partition(XM_PARTITION_SELF);
}
```

3.3 XAL partition programming

This section will show, step by step, the build process of the *Hello World* XAL partition example, task which is automated by the `make` utility.

The steps to build the final system image are the next:

1. Compilation of the partition code.
2. Compilation of the configuration file.
3. Build of the final system image.

3.3.1 Compilation of the partition code

The partition source code `partition.c` is compiled using the `gcc` compiler. The result is the compiled partition in XEF. The steps to achieve the compiled partition are the next (keep in mind that the *Hello World* example has two partitions with the same source code).

- **Step 1**: Compile the source code `partition.c` as follows:

```
$ make partition.o
...
```

- **Step 2**: Link the object `partition.o` to the fixed partition address. Notice that the example has two partitions with the same source code:

```
$ make partition0
...
$ make partition1
...
```

- **Step 3**: Build the partition XEF:

```
$ make partition0.xef
...
$ make partition1.xef
...
```

3.3.2 Compilation of the XtratuM XML configuration file

The XtratuM XML configuration file is compiled using the tool called `xmcparser`, which is distributed with the XM-SDK, and creates a compiled version in BIN format of the XML configuration file after the XML validation process.

- **Step 4**: Parse and compile the XtratuM XML configuration file:

```
$ make xm_cf.bin.xmc
```

3.3.3 Build of the final resident software

In this step the final system image is built taking as input the files that compose the system (hypervisor, partitions and their respective customization files, if any) as well as their final address location. The output is a bootloader with a container. This container contents the XEF files.

- **Step 5**: Pack the system in the final image file (`resident_sw`):

```
$ make resident_sw
```

The `Makefile` rule `resident_sw` uses the tool `xmpack` to create the container and the tool `rswbuild` to attach the container to the bootloader RSW. Theses tools are distributed with the XM-SDK.

3.4 XAL development process summary

The figure 3.1 summarizes the steps of the XAL partition build process:

- **Step 1**: Compile the C language sources (in this case, only one source file `hello.c`) with the `gcc` cross-compiler. The output will be the file `hello.o`.
- **Step 2**: Link the compiled object code at the partition address with the `ld` linker. In this example, the object code is linked to two partitions, `partition0` and `partition1`.
- **Step 3**: Build the partition XEF (XtratuM Executable Format) using the `xmeformat` tool.
- **Step 4**: Parse and compile the XtratuM XML configuration file with the `xmcparser` tool.
- **Step 5**: Pack all the system components in a bootloader (final system image). To achieve this step, `xmpack` and `rswbuild` tool is used.

All the required tools are distributed in the XM-SDK, except the compiler and the linker used in steps 1 and 2.

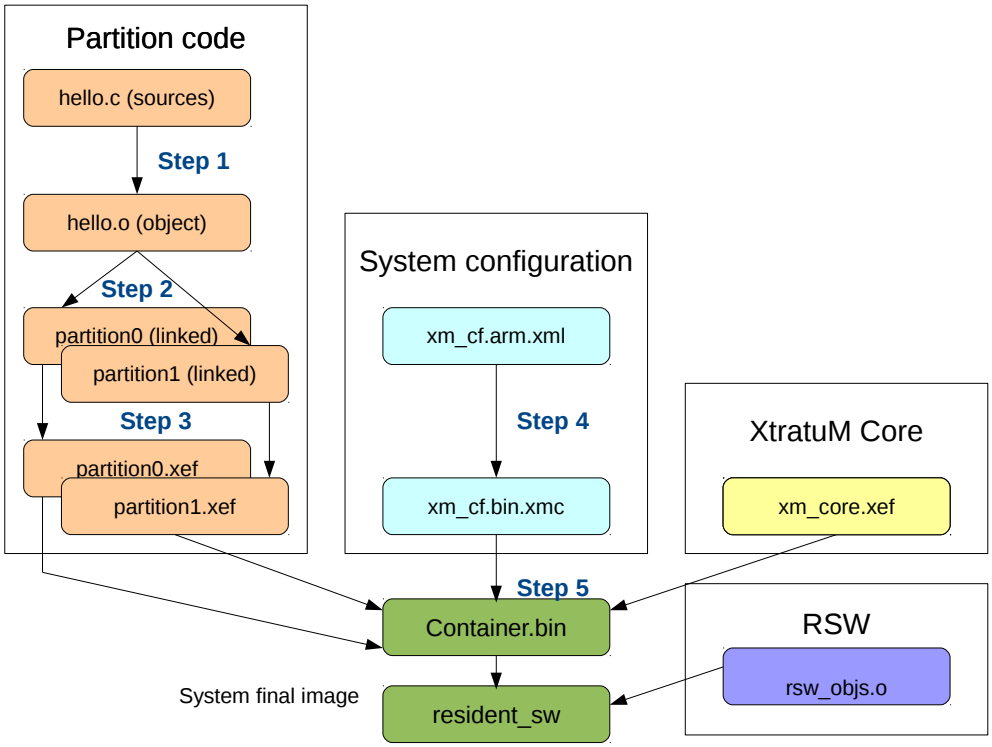

Figure 3.1: XAL development process summary.

This page is intentionally left blank.

Chapter 4

XtratuM system execution

The final system image packed in a single file (in the example called `resident_sw`) can be now loaded in the board, which is the ZEDBOARD from © Xilinx .

A software will be need in order to load the system image. This software call XMD, is part of a tool suit or IDE (Integrated Development Environment), called ISE, manufactured also by © Xilinx . The © Xilinx Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using Dual CORTEX-A9 MPCore processor. You can use it to debug programs CORTEX-A9 processors running on a hardware board, cycle-accurate Instruction Set Simulator (ISS).

This chapter is organized as follows:

- Section 4.1 refers briefly to the installation of ISE.
- Section 4.2 refers briefly the steps for ZEDBOARD hardware bring-up.
- Section 4.3 shows how to load, execute and debug a system image.

4.1 Installation

The needed tool XMD is provided as part of the Xilinx Design Tools. Concretely in the ISE Design Suite DVD.

The version tested in the development is 2014.2/2014.3 available for Linux, and provided with the board development tools DVD.

Please, refer to the DVD documentation or Xilinx's website to extra information about the installation and licensing process.

4.2 Board Bring Up.

This section details the hardware setup and use of the terminal program for running the Hello World XAL Example. It contains step-by-step instructions for ZEDBOARD bring-up.

Figure 4.1 provides an overview of the ZEDBOARD and its connections.

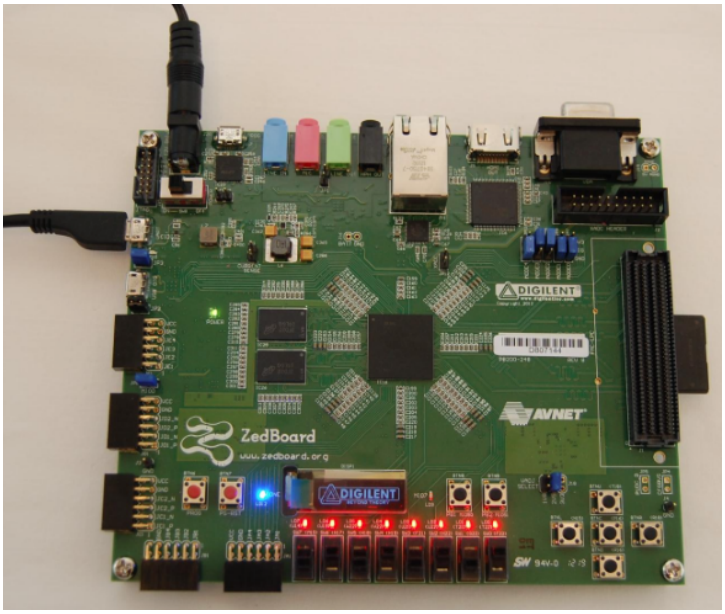

Figure 4.1: ZEDBOARD board features overview.

The steps to follow are:

1. Check if the board is switched off. If not unplug it.
2. Insert the 4GB SD card included with ZEDBOARD into the SD card slot (J12) located on the underside of ZEDBOARD PCB This SD card comes preloaded with demo software and contains a basic Linux configuration used to implement some demos. Figure 4.4.
3. Verify the Board boot mode (JP7-JP11) and MIO0 (JP6) jumpers are set to SD card mode as described in the Hardware Users Guide. Figure 4.3
4. Plug a MicroUSB cable into the UART port of the ZEDBOARD (J14) board and your control PC. Figure 4.2.
5. Plug a MicroUSB cable into the JTAG port of the ZEDBOARD (J17) board and your control PC. Figure 4.2.
6. Install the power cable 12V stand-alone power supply. (J20) Figure 4.5.
7. Turn power switch (SW8) to the ON position.

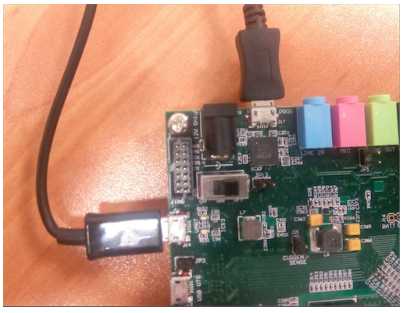

Figure 4.2: UART and DEBUG wires connection.

Figure 4.3: Boot mode jumpers.

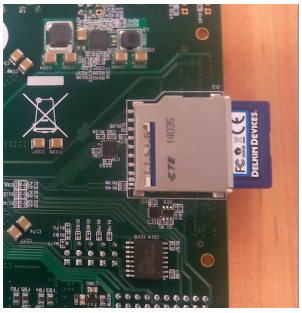

Figure 4.4: SD card place detail.

Figure 4.5: Power input connection.

4.3 Loading, execution and debugging

Once the system has been build, the resulting binary file must be burned/downloaded to the target board in order to execute it. In the following lines is explained how to perform it through the XMD.

The XMD console is a standard Tcl console, where you can run any available Tcl commands. Additionally, the XMD console provides command editing convenience, such as file and command name auto-fill and command history.

4.3.1 Launching the command line application

To launch the XMD in a terminal type $xdm. Then the application prompt the following lines:

```
$ xmd
Xilinx Microprocessor Debugger (XMD) Engine
Xilinx EDK 14.2 Build EDK_P.28xd
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.

XMD%
XMD%
```

If you have any problem finding the xmd command. Please, check that you have executed the command:

```
$source (PATH_XILINX)/SDK/(SDK_VERSION)/settings(ARCH).sh
```

4.3.2 Connecting to the board.

To connect the XMD tool to the board you need to plug an USB wire to the PROG slot. Then type in the XMD command line as could be observed:

```
XMD% connect arm hw

JTAG chain configuration
--------------------------------------------------
Device  ID Code       IR Length   Part Name
 1      4ba00477          4       arm_dap
 2      23727093          6       xc7z020

--------------------------------------------------
```

```
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".

-------------------------------------------------

CortexA9 Processor Configuration
-------------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........4

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
XMD%
```

If you find any problem with the board connection (USB-JTAG or USB-UART). Please, consider to reinstall the Xilinx driver ussing the script provided in the SDK:

```
$(PATH_XILINX)/SDK/(SDK_VERSION)/data/xicom/cable_drivers/lin(ARCH)/digilent/install_digilent.sh
```

4.3.3 Downloading the resident_sw int the board

Before downloading the binary is recomended to reset the debug system and the board using the following command:

```
XMD% rst -processor
Target reset successfully
0
XMD%
```

Then download the binary file follow the next command. The path to the file could be realtive, but we recomend to use absolute file system paths.

```
XMD% dow (PATH_TO_FILE)/resident_sw
Downloading Program -- ...../resident_sw
       section, .text: 0x3fe00000-0x3fe0237b
       section, .rodata: 0x3fe02380-0x3fe025bb
       section, .kbuild_info: 0x3fe025bc-0x3fe025e5
       section, .container: 0x3fe20000-0x3fe39153
       section, .data: 0x3fe10000-0x3fe10007
       section, .bss: 0x3fe10008-0x3fe1f43b
Setting PC with Program Start Address 0x3fe00cb0
XMD%
```

4.3.4 Catching the output. Uart Configuration

To see the output provided by the example, the UART must be properly configured. A valid configuration parameters set is provided in the Table 4.1:

Parameter	Value	Configurable by sw
Communication port	/dev/ttyACM0	No
Baud rate	115200	Yes
Data bits	8	Yes
Parity	None	Yes
Stop bits	1	Yes
Flow control	HW	No

Table 4.1: UART Valid Configuration.

```
       [RSW] Start Resident Software
       [RSW] Parse Container
       [RSW] Loading XM and XML config file
       [RSW] Checking xml config file
       [RSW] Load custom FilRSW] XEF load File
       [RSW] Loading additional custom files
       [RSW] Loading partitions
       [RSW] Loading partition 1
       [RSW] Loading partition 2
       [RSW] Starting XM at 0x20000000
       XM Hypervisor (2.0 r0) Built June 5 2015 10:03:28
       Detected 400.0MHz processor.
       >> HWClocks [CortexA9 Global Clock (1000Khz)]
       SetupHwTimer
       >> HwTimer [CortexA9 Private Timer1 (1000Khz)]
       2 Partition(s) created
       P0 ("Partition0":0) flags: [ SYSTEM ]:
       P1 ("Partition1":1) flags: [ SYSTEM ]:
           [0x14000000:0x14000000 - 0x1403ffff:0x1403ffff] flags: 0x0
       [P0] Hello World!
       [P1] Hello World!
```

Listing 4.1: Hello World Example output.

4.3.5 Run and Debug the application

Once the binary has been downloaded must be run by typing the next command:

```
XMD% run
Processor started. Type "stop" to stop processor

RUNNING> XMD%
```

Is recommended to perform a reset again each time the application is launched.

```
XMD% rst -srst
Successfully asserted SRST
JTAG chain configuration
-------------------------------------------------
Device  ID Code       IR Length   Part Name
1       4ba00477          4       arm_dap
```

```
2       23727093          6        xc7z020

Processor stopped

System reset successfully
XMD%
```

Once the processors are released, the uart will print the application output which should look like the following list 4.1

Additionally a quick way to download and run the application is to use TCL scripts. In most cases the command sequence to perform this is the same. These commands can be transcribed in a file as shown in list 4.2.

```
connect arm hw
rst -srst
dow resident_sw
run
```

Listing 4.2: xmd-loadNrun.tcl

In order to execute the tlc file, run the following command:

```
$ xmd -tcl xmd-loadNrun.tcl

****** Xilinx Microprocessor Debugger (XMD) Engine
****** XMD v2014.3 (64-bit)
  **** SW Build 1034051 on Fri Oct 3 16:32:26 MDT 2014
    ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

Executing user script : xmd-loadNrun.tcl

JTAG chain configuration
--------------------------------------------------
Device  ID Code       IR Length   Part Name
 1      4ba00477          4       arm_dap
 2      23731093          6       xc7z045

--------------------------------------------------
Enabling extended memory access checks for Zynq.
Writes to reserved memory are not permitted and reads return 0.
To disable this feature, run "debugconfig -memory_access_check disable".

--------------------------------------------------

CortexA9 Processor Configuration
-------------------------------------
Version.............................0x00000003
User ID.............................0x00000000
No of PC Breakpoints................6
No of Addr/Data Watchpoints.........4

Connected to "arm" target. id = 64
Starting GDB server for "arm" target (id = 64) at TCP port no 1234
Successfully asserted SRST
```

```
JTAG chain configuration
-------------------------------------------------
Device  ID Code       IR Length   Part Name
 1      4ba00477          4       arm_dap
 2      23731093          6       xc7z045

Processor stopped

System reset successfully

Processor Reset .... DONE
Downloading Program -- resident_sw
       section, .text: 0x00100000-0x00102443
       section, .rodata: 0x00102448-0x0010257b
       section, .kbuild_info: 0x0010257c-0x001025a5
       section, .container: 0x00300000-0x0033eeb7
       section, .data: 0x00200000-0x00200007
       section, .bss: 0x00200008-0x0020f43b
Download Progress..10.20.30.40.50.60.70.80.90..Done
Setting PC with Program Start Address 0x00100000
Processor started. Type "stop" to stop processor

RUNNING>
```

XMD provide an extensive set of debug commands. Information about the use of these commans could be obtained using the commands:

```
XMD% help

XMD Terminal Commands Types:
init......... Load/Initialize the System Files
connect...... Connect to System Target
files........ Load ELF/Data files
running...... Program Execution
breakpoints.. Setting Breakpoints/Watchpoints
trace........ Tracing and Profiling options
misc......... Miscellaneous Options
help......... Help on help

Type "help" to display XMD command types
Type "help" followed by above "type" for more options
XMD% help running

     Program Execution, Reading/Writing Registers and Memory
----- Syntax -----                    ----- Description -----
  run                                Run program from <Start Address>

  con [address]                      Continue program execution from
                                     current address
      [-block [-timeout <in secs>]]

  stp [num_instrns]                  Step
```

```
cstp [num_cycles]                    Cycle Step (Simulator/VP targets)

rst                                  Reset the System.
                                     For Zynq, get the active processor
                                     to trigger soft reset. This needs
                                     the system to be in a good state.
                                     HW breakpoints can be set through
                                     bps command before issuing reset.
                                     This is useful for debugging code
                                     booting from flash.

   [-processor]                      Reset the processor.

   [-slcr]                           Trigger SLCR reset on Zynq.

   [-debug_sys]                      Trigger Debug System Reset through
                                     DAP, on Zynq.

   [-srst]                           Assert the SRST pin on the JTAG
                                     cable to trigger soft reset on Zynq
                                         .

stop                                 Stop

rrd [reg num]                        Register Read

srrd [reg name]                      Special Register Read. For CortexA9
    ,
                                     read a set of CoProcessor regs,
                                     identified by [reg name].
                                     [reg name] can be any of ctrl,
                                     debug, dma, tcm, id, etc, vfp.
                                     (default: ctrl)

mrc <CPx> <op1> <CRn> <CRm> <op2>    ARM CoProcessor Register Read

dp_rrd <reg offset>                  ARM Debug Port Register Read

ap_rrd <reg offset>                  ARM Access Port Register Read

rwr <register> <word>                General purpose Register Write

mcr <CPx> <op1> <CRn> <CRm> <op2> <word> ARM CoProcessor Register Write

dp_rwr <reg offset>                  ARM Debug Port Register Write

ap_rwr <reg offset>                  ARM Access Port Register Write

mrd <address> [num] [w|h|b]          Memory Read 'w'ord/'h'alfword/'b'
    yte
                                     (default: w)

mrd_var <variable name> [ELF file]   Read memory at global variable
```

```
  mrd_phys <address> [num] [w|h|b]      ARM Memory Read through AHB AP
                                        (default: 'w'ord)
                   [-force]             Read from OCM at 0x0 (iff DDR is
                       not
                                        remapped to 0x0)

  dmrd <address>                        Read CoreSight Memory

  mwr <address> <values> [<num> <w|h|b>] Memory Write (default: 'w'ord)

  mwr_phys <addr> <values> [<num> <w|h|b>] ARM Memory Write through AHB AP
                                        (default: 'w'ord)
                            [-force]    Write to OCM at 0x0 (iff DDR is not
                                        remapped to 0x0)

  dmwr <address>                        Write to CoreSight Memory

  safemode [-config mode <exception_mask>] Enable/disable & configure
      safemode.

          [-config <exception_id> <exception_address>]
                                        Change exception handler addresses

          [on|off]                      Enable/disable safemode.
          [-info]                       Display current safemode
                                        configuration.
          [-elf <elf_file>]             Specify the elf file to be debugged
              .

XMD%
```

This page is intentionally left blank.

Bibliography

[1] Fent Innovative Software Solutions. XM-ARM: Software reference manual. Technical Report 14-035-03.006.sum.01, Fent Innovative Software Solutions, S.L., June, 2015.

[2] Fent Innovative Software Solutions. XM-ARM: Software user manual. Technical Report 14-035-03.005.sum.01, Fent Innovative Software Solutions, S.L., June, 2015.

This page is intentionally left blank.

Appendix A

Known Issues and Problems.

A.1 «xmd» command not found

A message of Command not found indicates one of the following thing:

- The xmd command is not available on the system. (It's assumed that is not your case if you install the Xilinx SDK).
- The xmd command is not in the search path.

To fix the search path problem, you need to source the file that Xilinx provide to this purpose, as is shown bottom:

```
$source (PATH_XILINX)/SDK/(SDK_VERSION)/settings(ARCH).sh
```

As additional advice, we recommend to add this in your shell login file, to avoid to do it each time you open a new shell. You can use the following command to perform it:

```
$source (PATH_XILINX)/SDK/(SDK_VERSION)/settings(ARCH).sh >> /home/<user
    >/.bashrc
```