ENCLUSTRA
FPGA SOLUTIONS

Enclustra Build Environment - User Documentation

Antmicro Ltd for Enclustra GmbH

2015-11-23

1 Introduction **1**
1.1 Version Information 1

2 Build Environment **2**
2.1 Prerequisites 2
2.2 Directory Structure 4
2.3 Repositories Structure 4
2.4 General Build Environment Configuration 5

3 Supported Devices **6**

4 Usage **7**
4.1 GUI 7
4.2 Command Line 11

5 Deployment **14**
5.1 SD Card (MMC) 15
5.2 QSPI Flash 16
5.3 NAND Flash 17
5.4 USB Drive 19
5.5 NFS 19

6 FAQ **21**
6.1 How to script U-Boot? 21
6.2 How can the flash memory be programmed from Linux? 22

CHAPTER

ONE

INTRODUCTION

This is the user documentation for the Enclustra Build Environment project.

1.1 Version Information

Date	Rev	Author	Changes
2015-05-08	0.1.0	Karol Gugala	Builsystem description
2015-05-11	0.1.1	Aleksandra Szawara	Language check
2015-07-06	0.1.2	Aurelio Lucchesi	Minor corrections
2015-11-20	0.1.3	Tomasz Gorochowik	Major reorganization

CHAPTER

TWO

BUILD ENVIRONMENT

This chapter describes the usage of the build environment. The whole build environment is written in `Python`. Its internal functionality is determined by *ini* files placed in a specific directory layout.

2.1 Prerequisites

To run the build script a Python interpreter is required. The system is compatible with both, Python 2 and Python 3.

The build environment requires additional software to be installed as listed below:

Table 2.1: Required software

tool	minimal version	comments
dialog	1.1-20120215	Required only in the GUI mode
make	3.79.1	
git	1.7.8	
tar	1.15	
wget	1.0	
c++ compiler		Required to build a busybox rootfs
gcc		Required to build the Linux kernel, U-Boot and a busybox rootfs

For more information on how to install the required packages in the supported systems, please refer to the corresponding subsection (*OpenSUSE 13.2 (Harlequin)*, *CentOS 7*, *Ubuntu 14.04 LTS*).

Additionally, the following Python modules are required (this applies to every supported distribution):

- os2emxpath
- backports
- ntpath
- pkg_resources
- opcode

- posixpath
- sre_constants
- nturl2path
- sre_parse
- sre_compile
- pyexpat
- strop
- genericpath
- repr

Those packages can be obtained by using `pip`:

```
sudo pip install os2emxpath backports ntpath pkg_resources opcode posixpath sre_constants\
                 nturl2path sre_parse sre_compile pyexpat strop genericpath repr
```

Note: Either the `dialog` Python module or the external `dialog` application is required to use the build environment's GUI.

2.1.1 OpenSUSE 13.2 (Harlequin)

```
sudo pip install argparse
sudo yzpper install -y dialog git make
sudo yzpper install -y u-boot-tools gcc patch
sudo yzpper install -y gcc-c++
sudo yzpper install -y flex bison
sudo yzpper install -y linux32
```

2.1.2 CentOS 7

```
sudo yum install -y dialog make git tar wget
sudo yum -y groupinstall 'Development Tools'
sudo yum install -y glibc.i686 libgcc.i686 libstdc++.i686 glibc-devel.i686
```

2.1.3 Ubuntu 14.04 LTS

```
sudo apt-get install -y u-boot-tools
sudo apt-get install -y git
sudo apt-get install -y gcc-multilib
sudo apt-get install -y lib32stdc++6
sudo apt-get install -y python-pip python-dev build-essential
sudo pip install --upgrade pip
sudo pip install --upgrade virtualenv
```

2.2 Directory Structure

The build environment is designed to work with a specific directory structure depicted below:

```
|-- bin
|-- binaries
|-- buildscripts
|-- sources
|   |-- target_submodule_1
|   |-- target_submodule_2
|   |-- target_submodule_3
|   |-- target_submodule_4
|-- targets
|   |-- Board_1
|       |-- Module_1
|       |-- Module_2
|   |-- Board_2
|       |-- Module_1
|-- target_output
```

Table 2.2: Folder description

Folder	function
bin	Remote toolchains installation folder.
binaries	Additional target binaries download folder.
sources	master_git_repository clone folder. It contains submodule folders.
build-scripts	Build system executable files.
targets	Target configurations are placed here.
target_output	Folders generated during the build process, that contain the output files after a successful build of every specifc target.

Important: By default, the output folders are named according to this folder naming scheme: `out_<timestamp>_<module>_<board>_<bootmode>`.

The default name can be overwriten during the build process.

2.3 Repositories Structure

The `sources` directory is the master git repository with a number of submodules pointing to actual code repositories. During the fetch phase, the build environment synchronizes only the submodules required to build the selected targets.

```
.
|-- container_git_repository
|   |-- target_submodule_1
|   |-- target_submodule_2
|   |-- target_submodule_3
```

2.4 General Build Environment Configuration

Environment settings are stored in the `enclustra.ini` file in the main directory of the build environment. Before starting the build script, one may need to adjust the general settings of the build environment by editing this file. One of the most crucial setting is the number of build threads used in a parallel. This parameter is set in the `[general]` section by changing the `ntreads` key. Additionally, parameters in the `[debug]` section allow the user to adjust the logging settings:

- If the `debug-calls` option if set to `true`, the output of all external tool calls (such as `make`, `tar` etc.) will be displayed in the terminal.
- If the `quiet-mode` option is set to `true`, the build log of the targets will not be printed to the terminal, only informations about actual build state will be shown. This option does not affect the `build-logfile` option.
- If the `build-logfile` option is set to a file name, the build environment will write the whole build log output to that file. If the option is not set, the output will not be logged.
- If the `break-on-error` option is set to `true`, the build environment will interrupted on the first error. Otherwise the build environment will only print an error message and continue to work on a next available target.

CHAPTER

THREE

SUPPORTED DEVICES

Table 3.1: Supported devices

Family	Module	Base board	Available targets
Xilinx	Mars ZX2	Mars Starter	Linux, U-Boot, Busybox
Xilinx	Mars ZX2	Mars EB1	Linux, U-Boot, Busybox
Xilinx	Mars ZX2	Mars PM3	Linux, U-Boot, Busybox
Xilinx	Mars ZX3	Mars Starter	Linux, U-Boot, Busybox
Xilinx	Mars ZX3	Mars EB1	Linux, U-Boot, Busybox
Xilinx	Mars ZX3	Mars PM3	Linux, U-Boot, Busybox
Xilinx	Mercury ZX1	Mercury PE1	Linux, U-Boot, Busybox
Xilinx	Mercury ZX5	Mercury PE1	Linux, U-Boot, Busybox

CHAPTER

FOUR

USAGE

4.1 GUI

In order to build the software for a chosen board using the GUI, please follow these steps:

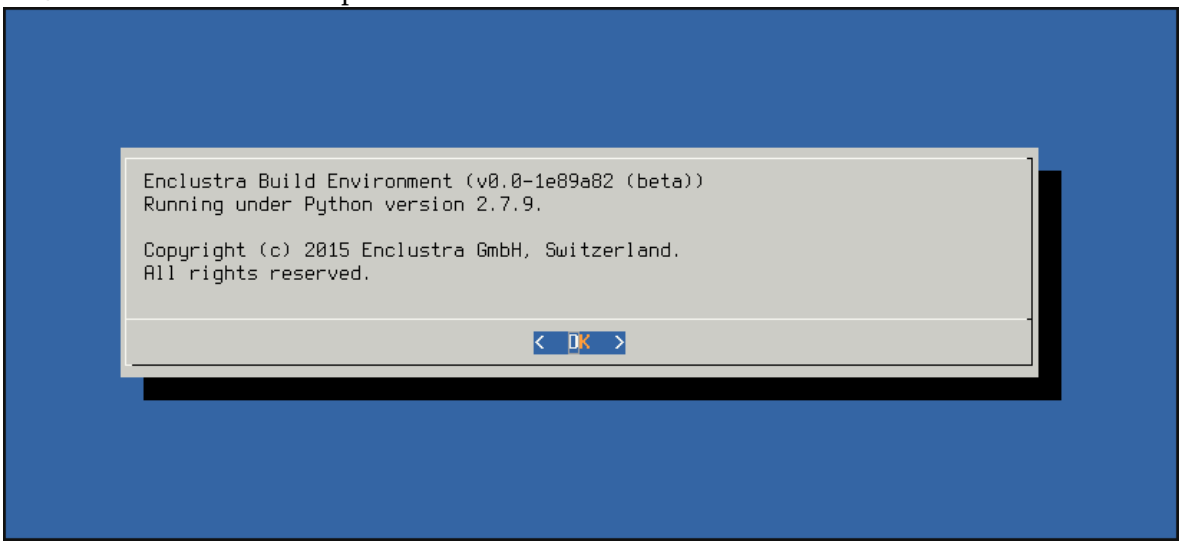
1. Clone the build environment repository with:

```
git clone https://github.com/enclustra-bsp/bsp-xilinx.git
```

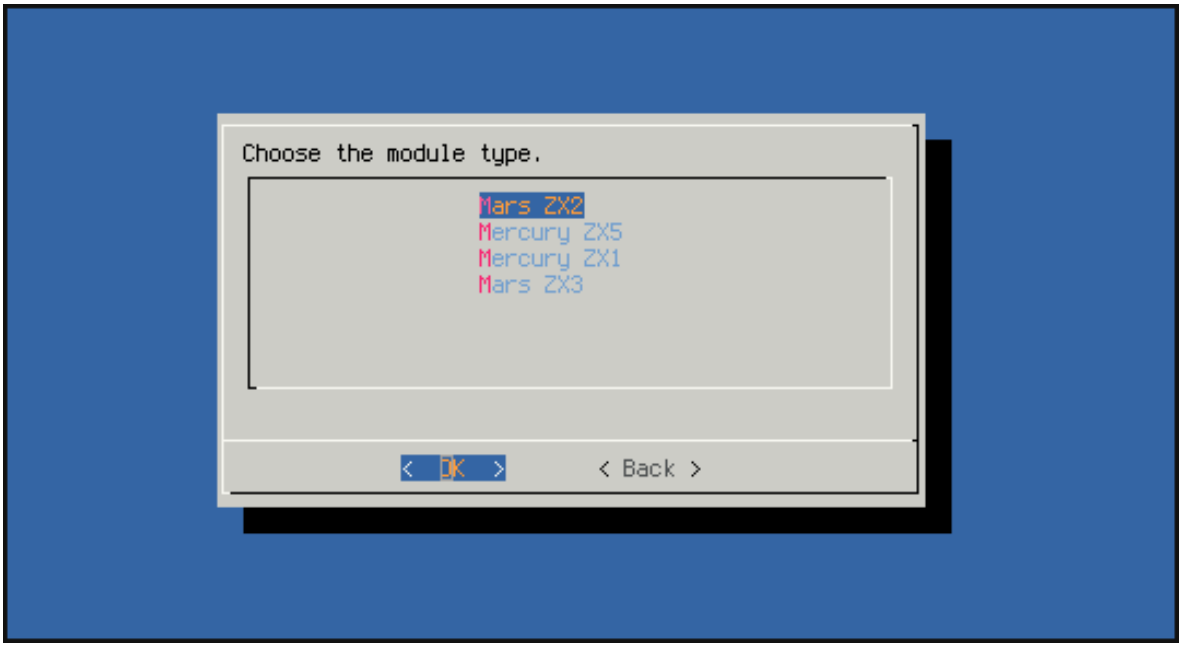
2. Change to the enclustra-buildscripts directory:

```
cd enclustra-buildscripts
```

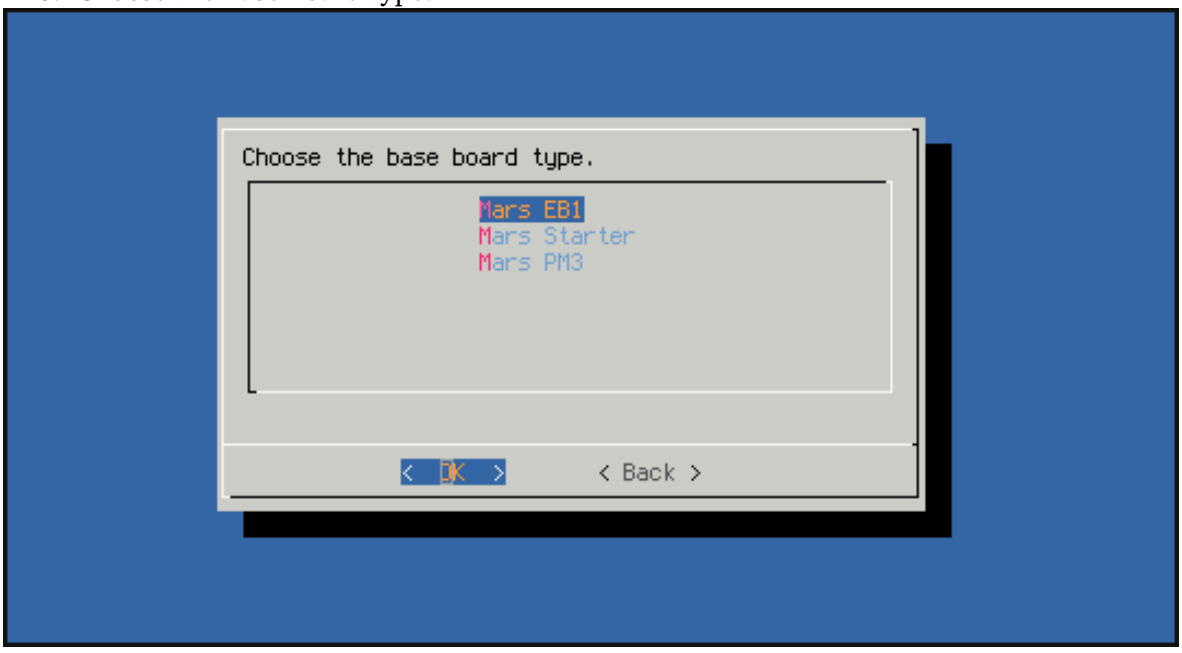
3. Run `./build.sh` script.
4. The welcome screen provides basic information about the version of the build environment.



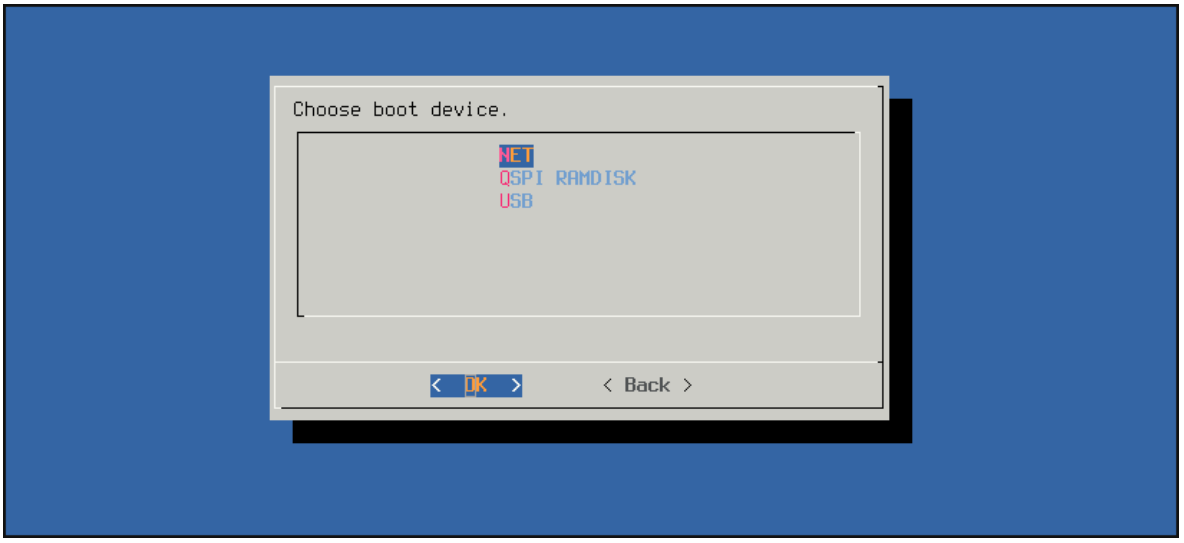
5. Choose the module type.



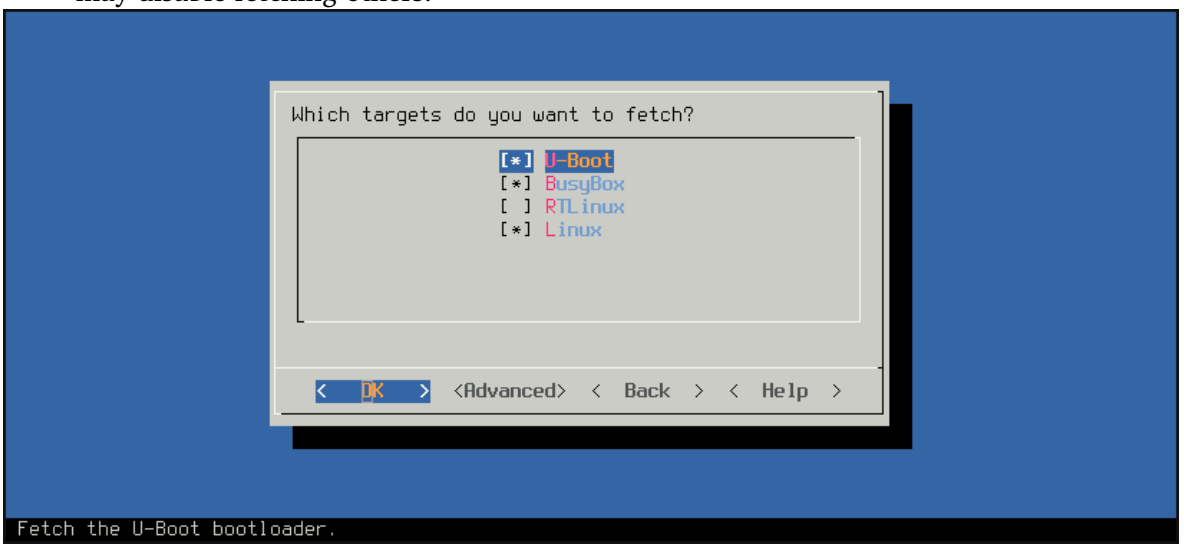
6. Choose the base board type.



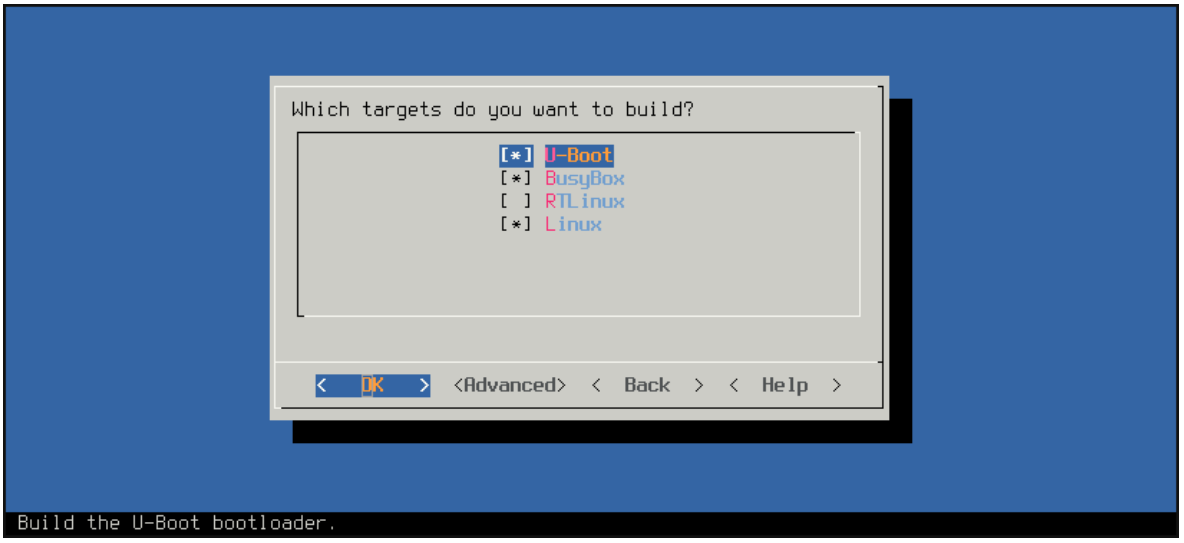
7. Choose the boot device.



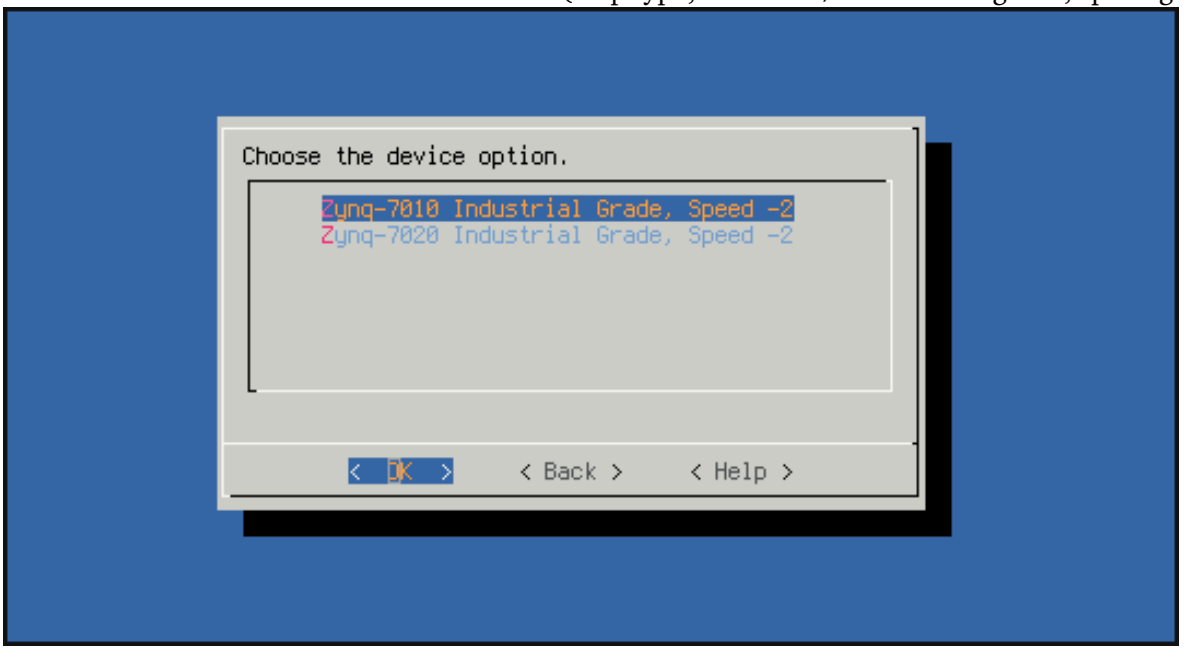
8. Choose which targets available for the chosen device family will be fetched. On the bottom of the screen a short description of the highlighted target is displayed. Choosing certain targets may disable fetching others.



9. Choose which targets will be built. On the bottom of the screen a short description of a highlighted target is displayed. Choosing certain targets may disable building others.

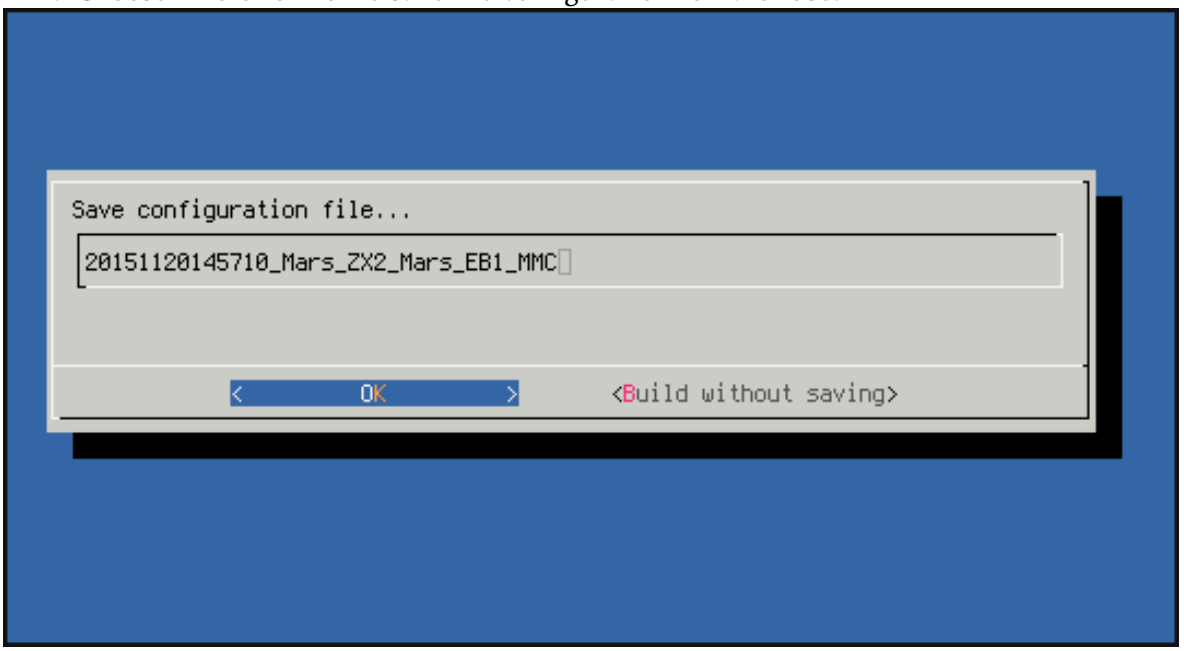


10. Choose the exact version of the device (chip type, industrial/commercial grade, speed grade).



11. Verify all the chosen build parameters.

12. Choose wheter or not to save the configuration for later use.



13. The build environment will fetch and build the chosen targets.

4.2 Command Line

The build process can be invoked from the command line. All options that are available in the GUI are present on the command line interface as well. A list of the available command line options can be obtained like this:

```
./build.sh --help

usage: tool [-h] [-L] [-d device] [-l] [-x target] [-f target] [-b target]
            [--fetch-history target] [--list-dev-options] [-o index] [-c] [-v]

Enclustra Build Environment

optional arguments:
  -h, --help                    show this help message and exit
  -L, --list-devices            list all available devices
  -d device, --device device    specify device as follows:
                                <module>/<base_board>/<boot_device>
  -l, --list-targets            list all targets for chosen device
  -x target                     fetch and build specific target
  -f target, --fetch target     fetch specific target
  -b target, --build target     build specific target
  --fetch-history target        fetch specific target with history
  --list-dev-options            list all available device options for chosen
                                device
  -o index, --dev-option index  set device option by index, the default one
                                will be used if not specified
  -c, --clean-all               delete all downloaded code, binaries, tools
                                and built files
  -v, --version                 print version
```

In order to list all available devices use the following command:

```
./build.sh -L
```

If the `build.sh` script is invoked with the `-d` option, the build environment switches to console mode. This mode requires a valid device specifier in order to locate the device configuration within the `targets` directory for the specific device, e.g. for the *Mars ZX3* module on the *Mars PM3* base board in *QSPI* boot mode, the command would look like this:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI
```

Such a command will fetch and build all the default targets for a selected device. To list all the available targets for a selected device, the user needs to add the `-l` switch to the command, e.g.:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI -l
```

The `-x` option will fetch and build only the selected target, e.g.:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI -x Linux
```

That will fetch and build only the `Linux` target for the selected device.

To only fetch or build a specific target, the user can specify those targets with the `-f` (fetch) and `-b` (build) options. It is possible to choose multiple targets, e.g. like this:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI -f Linux -b BusyBox -x U-Boot
```

This will fetch Linux, build BusyBox and fetch/build U-Boot.

The `--list-dev-options` option will list all the available device options for the chosen device. It can be used like this:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI -x Linux --list-dev-options
```

This will print out an indexed list of device options.

The `-o` option allows the user to choose a device option for the selected device by providing the index of a specific device option, like this:

```
./build.sh -d Mars_ZX3/Mars_PM3/QSPI -x Linux -o 1
```

If no device option is selected, the default one will be used.

To reset the build environment and delete all downloaded code, binaries, tools and built files, the `--clean` option can be used:

```
./build.sh --clean
```

CHAPTER

FIVE

DEPLOYMENT

This chapter describes how to prepare the hardware to boot from different boot media, using the binaries generated from the build environment. The boot process differs in its details on different hardware, but in general it covers the following steps:

1. BootRom embedded in a CPU starts execution after reset. It searches through a predefined storages for a next (first) stage boot loader.
2. First stage boot loader (FSBL, U-Boot SPL) is loaded into On Chip Memory, and executed.
3. First stage boot loader initializes RAM controller, clocks and loads second stage boot loader into RAM.
4. Second stage boot loader (U-Boot) loads the Linux kernel, device tree blob and any other required files into RAM and runs the Linux kernel.
5. Linux kernel configures peripherals, mounts user space root filesystem and executes the `init` application within it.
6. `Init` application starts the rest of the user space applications - the system is up and running.

For more detailed information about the boot process on a Xilinx Zynq devices please refer to:

- Xilinx Zynq technical reference guide (chapters 6 and 32).

All the guides in this section require the user to build the required files for chosen device, with the build environment, like described in the previous section. After building the files, they can be deployed to the hardware, like described in the following sub sections.

Note: Default target output folders are named according to this folder naming scheme:

`out_<timestamp>_<module>_<board>_<bootmode>`.

As a general note on U-Boot used in all the following guides: U-Boot is using variables from the default environment. Moreover, the boot scripts used by U-Boot also rely on those variables. - If the environment was changed and saved earlier, U-Boot will always use these saved environment variables on a fresh boot, even after changing the U-Boot environment. To restore the default environment, run the following command in the U-Boot command line:

```
env default -a
```

This will not overwrite the stored environment but will only restore the default one in the current run. To permanently restore the default environment, the `saveenv` command has to be invoked.

Note: A warning like that `*** Warning - bad CRC, using default environment` when booting into U-Boot indicates that the default environment will be loaded.

5.1 SD Card (MMC)

In order to deploy images to an SD Card and boot from it, do the following steps:

1. Create a FAT formatted BOOT partition as the first one on a SD Card. The size of the partition should be at least 16 MB. (For more information on how to prepare the boot medium, please refer to the official Xilinx guide.)
2. Create an ext2 formatted partition (rootfs) as the second one on a SD Card. The size of the partition should be at least 16 MB.
3. Copy `boot.bin`, `uImage`, `devicetree.dtb` and `uboot.scr` from the build environment output directory onto the BOOT partition (FAT formatted).
4. Extract the `rootfs.tar` archive from the build environment output directory onto the second partition (rootfs, ext2 formatted). This must be done as a root.

```
sudo tar -xpf rootfs.tar -C /path/to/mmc/mountpoint
```

5. Unmount all partitions mounted from the SD Card.
6. Insert the card into the SD Card slot on the board.
7. Configure the board to boot from the SD Card (refer to the board User Manual).
8. Power on the board.
9. The board should boot the Linux system.

If one wants to manually trigger booting from a SD Card, the following command has to be invoked from the U-Boot command line:

```
run sdboot
```

5.2 QSPI Flash

Table 5.1: Xilinx Family QSPI Flash Layout

Partition	Offset	Size
Boot image	0x0	0x600000
Linux kernel	0x600000	0x500000
Linux Device Tree	0xB00000	0x80000
U-Boot environment	0xB08000	0x80000
Bootscript	0xC00000	0x80000
Rootfs	0xC40000	0x3C0000

In order to deploy images to QSPI Flash and boot from it, do the following steps:

1. Setup a TFTP server on the host computer.
2. Power on the board and boot to U-Boot (e.g. from a SD Card (MMC)).
3. Connect an Ethernet cable to the device.
4. Connect a serial console to the device (e.g. using PuTTY or picocom).
5. Setup the U-Boot connection parameters (in the U-Boot console):

```
setenv ipaddr 'xxx.xxx.xxx.xxx'
# where xxx.xxx.xxx.xxx is the board address
setenv serverip 'yyy.yyy.yyy.yyy'
# where yyy.yyy.yyy.yyy is the server (host computer) address
```

6. Copy `boot.bin`, `uImage`, `devicetree.dtb`, `uboot.scr` and `uramdisk` from the build environment output directory to the TFTP server directory
7. Set memory pinmux to QSPI Flash:

```
zx_set_storage QSPI
```

8. Update the boot image:

```
mw.b ${bootimage_loadaddr} 0xFF ${bootimage_size}
tftpboot ${bootimage_loadaddr} ${bootimage_image}
sf probe
sf erase ${qspi_bootimage_offset} ${bootimage_size}
sf write ${bootimage_loadaddr} ${qspi_bootimage_offset} ${filesize}
```

9. Update the boot script image:

```
mw.b ${bootscript_loadaddr} 0xFF ${bootscript_size}
tftpboot ${bootscript_loadaddr} ${bootscript_image}
sf probe
sf erase ${qspi_bootscript_offset} ${bootscript_size}
sf write ${bootscript_loadaddr} ${qspi_bootscript_offset} ${filesize}
```

10. Update the Linux kernel:

```
mw.b ${kernel_loadaddr} 0xFF ${kernel_size}
tftpboot ${kernel_loadaddr} ${kernel_image}
sf probe
sf erase ${qspi_kernel_offset} ${kernel_size}
sf write ${kernel_loadaddr} ${qspi_kernel_offset} ${filesize}
```

11. Update the devicetree image:

```
mw.b ${devicetree_loadaddr} 0xFF ${devicetree_size}
tftpboot ${devicetree_loadaddr} ${devicetree_image}
sf probe
sf erase ${qspi_devicetree_offset} ${devicetree_size}
sf write ${devicetree_loadaddr} ${qspi_devicetree_offset} ${filesize}
```

12. Update the rootfs image:

```
mw.b ${ramdisk_loadaddr} 0xFF ${ramdisk_size}
tftpboot ${ramdisk_loadaddr} ${ramdisk_image}
sf probe
sf erase ${qspi_ramdisk_offset} ${ramdisk_size}
sf write ${ramdisk_loadaddr} ${qspi_ramdisk_offset} ${filesize}
```

13. Power off the board.
14. Configure the board to boot from the QSPI Flash (refer to the board User Manual).
15. Power on the board.
16. The board should boot the Linux system.

If one wants to manually trigger booting from the QSPI Flash, the following command has to be invoked from the U-Boot command line:

```
run qspiboot
```

Note: Note that step 8 to 12 can be invoked independently.

5.3 NAND Flash

The Xilinx family devices cannot boot directly from a NAND Flash memory. The FSBL and the U-Boot have to be started from SD Card (MMC) or QSPI Flash. Please refer to *SD Card (MMC)* or *QSPI Flash* in order to boot U-Boot from SD Card or QSPI Flash. When U-Boot is booted it can load and boot the Linux system stored on the NAND Flash memory.

Table 5.2: Xilinx Family NAND Flash Layout

Partition	Offset	Size
Linux kernel	0x0	0x500000
Linux Device Tree	0x500000	0x100000
Bootscript	0x600000	0x100000
Rootfs	0x700000	Rest of the NAND Storage space

Note: Not all Xilinx-based modules come with NAND Flash memory.

In order to deploy images and boot the Linux system from NAND Flash, do the following steps:

1. Setup an TFTP server on the host computer.
2. Power on the board and boot to U-Boot (e.g. from a SD Card (MMC)).
3. Connect an Ethernet cable to the device.
4. Connect a serial console to the device (e.g. using PuTTY or picocom).
5. Copy `uImage`, `devicetree.dtb`, `uboot.scr` and `rootfs.ubi` files from the build environment output directory to the TFTP server directory.
6. Setup the U-Boot connection parameters (in the U-Boot console):

```
setenv ipaddr 'xxx.xxx.xxx.xxx'
# where xxx.xxx.xxx.xxx is the board address
setenv serverip 'yyy.yyy.yyy.yyy'
# where yyy.yyy.yyy.yyy is the server (host computer) address
```

7. Stop the U-Boot autoboot.
8. Set the memory pinmux to NAND Flash:

```
zx_set_storage NAND
```

9. Update the boot script image:

```
mw.b ${bootscript_loadaddr} 0xFF ${bootscript_size}
tftpboot ${bootscript_loadaddr} ${bootscript_image}
nand device 0
nand erase.part nand-bootscript
nand write ${bootscript_loadaddr} nand-bootscript ${filesize}
```

10. Update the Linux kernel:

```
mw.b ${kernel_loadaddr} 0xFF ${kernel_size}
tftpboot ${kernel_loadaddr} ${kernel_image}
nand device 0
nand erase.part nand-linux
nand write ${kernel_loadaddr} nand-linux ${filesize}
```

11. Update the devicetree image:

```
mw.b ${devicetree_loadaddr} 0xFF ${devicetree_size}
tftpboot ${devicetree_loadaddr} ${devicetree_image}
nand device 0
nand erase.part nand-device-tree
nand write ${devicetree_loadaddr} nand-device-tree ${filesize}
```

12. Update the rootfs image:

```
mw.b ${ubifs_loadaddr} 0xFF ${ubifs_size}
tftpboot ${ubifs_loadaddr} ${ubifs_image}
nand device 0
nand erase.part nand-rootfs
nand write ${ubifs_loadaddr} nand-rootfs ${filesize}
```

13. Trigger NAND Flash boot with:

```
run nandboot
```

Note: Note that step 8 to 11 can be invoked independently.

5.4 USB Drive

The Xilinx family devices cannot boot directly from a USB Drive. The FSBL and the U-Boot have to be started from SD Card (MMC) or QSPI Flash. Please refer to *SD Card (MMC)* or *QSPI Flash* in order to boot U-Boot from SD Card or QSPI Flash. When U-Boot is booted it can load and boot the Linux system stored on the USB Drive.

In order to deploy images and boot the Linux system from a USB Drive, do the following steps:

1. Create a FAT formatted partition as the first partition on the drive. This partition should have at least 16 MiB. (For more information on how to prepare the boot medium, please refer to the official Xilinx guide.)
2. Copy `uImage`, `devicetree.dtb`, `uramdisk` and `uboot.scr` from the build environment output directory to the FAT formatted partition.
3. Insert the USB drive into the USB port of the board.
4. Configure the board to boot from the SD Card (MMC) or QSPI Flash (refer to the board User Manual).
5. Power on the board and stop the U-Boot autoboot.
6. Trigger USB boot with:

```
run usbboot
```

5.5 NFS

The Xilinx family devices cannot boot directly from NFS.

The FSBL and the U-Boot have to be started from SD Card (MMC), with the images generated by the build environment. When U-Boot is booted it can load and boot the Linux system from the host machine via Ethernet. Please refer to nfs_preparation_guide to prepare your system for NFS boot.

In order to deploy images and boot the Linux system over NFS, do the following steps:

1. Create a FAT formatted BOOT partition as the first one on a SD Card. The size of the partition should be at least 16 MB. (For more information on how to prepare the boot medium, please refer to the official Xilinx guide.)
2. Copy `boot.bin` and `uboot.scr` from the build environment output directory onto the BOOT partition.
3. Extract the `rootfs.tar` archive from the build environment output directory into the NFS folder. This must be done as a root.

```
sudo tar -xpf rootfs.tar -C /path/to/NFS
```

4. Copy the `uImage`, `devicetree.dtb` and `uboot.scr` to the TFTP folder.
5. Insert the card into the SD Card slot on the board.
6. Configure the board to boot from the SD Card (MMC).
7. Power on the board and stop the U-Boot autoboot.
8. Set the server's and target's IP address, and the path to the rootfs NFS folder.

```
env default -a
setenv ipaddr 192.168.1.10
setenv serverip 192.168.1.2
setenv serverpath /path/to/NFS
saveenv
```

9. Trigger NFS boot with:

```
run netboot
```

Note: Saving the U-Boot environment like above ensures that NFS boot will work automatically after reboot.

CHAPTER

SIX

FAQ

6.1 How to script U-Boot?

All U-Boot commands can be automated by scripting, so that it is much more convenient to deploy flash images to the hardware.

For example, QSPI deployment:

Put the following commands as plain text to a file `cmd.txt`:

```
mw.b ${bootimage_loadaddr} 0xFF ${bootimage_size}
tftpboot ${bootimage_loadaddr} ${bootimage_image}
sf probe
sf erase ${qspi_bootimage_offset} ${bootimage_size}
sf write ${bootimage_loadaddr} ${qspi_bootimage_offset} ${filesize}

mw.b ${bootscript_loadaddr} 0xFF ${bootscript_size}
tftpboot ${bootscript_loadaddr} ${bootscript_image}
sf probe
sf erase ${qspi_bootscript_offset} ${bootscript_size}
sf write ${bootscript_loadaddr} ${qspi_bootscript_offset} ${filesize}

mw.b ${kernel_loadaddr} 0xFF ${kernel_size}
tftpboot ${kernel_loadaddr} ${kernel_image}
sf probe
sf erase ${qspi_kernel_offset} ${kernel_size}
sf write ${kernel_loadaddr} ${qspi_kernel_offset} ${filesize}

mw.b ${devicetree_loadaddr} 0xFF ${devicetree_size}
tftpboot ${devicetree_loadaddr} ${devicetree_image}
sf probe
sf erase ${qspi_devicetree_offset} ${devicetree_size}
sf write ${devicetree_loadaddr} ${qspi_devicetree_offset} ${filesize}

mw.b ${ramdisk_loadaddr} 0xFF ${ramdisk_size}
tftpboot ${ramdisk_loadaddr} ${ramdisk_image}
sf probe
sf erase ${qspi_ramdisk_offset} ${ramdisk_size}
sf write ${ramdisk_loadaddr} ${qspi_ramdisk_offset} ${filesize}

run qspiboot
```

Then generate an image `cmd.img` and put it onto the TFTP server on the host computer like this:

```
mkimage -T script -C none -n "QSPI flash commands" -d cmd.txt cmd.img
cp cmd.img /tftpboot
```

And finally, load the file on the target platform in U-boot and execute it, like this (after step 5 Setup U-Boot connection parameters, in the user documentation):

```
tftpboot 100000 cmd.img
source 100000
```

6.2 How can the flash memory be programmed from Linux?

In order to program flash memory from Linux, a script like the following one can be used. - All required files need to be present in the current folder. They can be loaded via TFTP or from USB drive / SD card.

```
#!/bin/sh

getsize ()
{
        local  size=`ls -al $1 | awk '{ print $5 }'`
        echo "$size"
}

PRELOADER_FILE="preloader-file"
PRELOADER_OFFSET="0"
UBOOT_FILE="u-boot-file"
UBOOT_OFFSET="0x60000"
BITSTREAM_FILE="fpga-file"
BITSTREAM_OFFSET="0x100000"
SCRIPT_FILE="uboot.scr"
SCRIPT_OFFSET="0x880000"
DEVICETREE_FILE="devicetree.dtb"
DEVICETREE_OFFSET="0x840000"
KERNEL_FILE="uImage"
KERNEL_OFFSET="0x8C0000"
ROOTFS_FILE="rootfs.jffs2"
ROOTFS_OFFSET="0"

flash_erase /dev/mtd0 0 0
FILESIZE=`getsize ${PRELOADER_FILE}`
echo Writing preloader file ${PRELOADER_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${PRELOADER_OFFSET} ${FILESIZE} ${PRELOADER_FILE}

FILESIZE=`getsize ${UBOOT_FILE}`
echo Writing uboot file ${UBOOT_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${UBOOT_OFFSET} ${FILESIZE} ${UBOOT_FILE}

FILESIZE=`getsize ${BITSTREAM_FILE}`
echo Writing bitstream file ${BITSTREAM_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${BITSTREAM_OFFSET} ${FILESIZE} ${BITSTREAM_FILE}
```

```
FILESIZE=`getsize ${SCRIPT_FILE}`
echo Writing bootscript file ${SCRIPT_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${SCRIPT_OFFSET} ${FILESIZE} ${SCRIPT_FILE}

FILESIZE=`getsize ${DEVICETREE_FILE}`
echo Writing devicetree ${DEVICETREE_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${DEVICETREE_OFFSET} ${FILESIZE} ${DEVICETREE_FILE}

FILESIZE=`getsize ${KERNEL_FILE}`
echo Writing kernel file ${KERNEL_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd0 ${KERNEL_OFFSET} ${FILESIZE} ${KERNEL_FILE}

flash_erase /dev/mtd1 0 0
FILESIZE=`getsize ${ROOTFS_FILE}`
echo Writing rootfs file ${ROOTFS_FILE} size ${FILESIZE}
mtd_debug write /dev/mtd1 ${ROOTFS_OFFSET} ${FILESIZE} ${ROOTFS_FILE}
```

Just make the script executable and execute it like this:

```
chmod +x flash.sh
./flash.sh
```

Note: Please refer to the user documentation of the developer tools for more information.