Enclustra Build Environment - User Documentation

Antmicro Ltd for Enclustra GmbH

2015-11-23

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

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.
tar-get_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.

SUPPORTED DEVICES

Table 3.1: Supported devices

Family	Module	Base board	Available targets
Altera	Mercury SA1	Mercury PE1	Linux, U-Boot, Busybox
Altera	Mercury SA2	Mercury PE1	Linux, U-Boot, Busybox

USAGE

4.1 GUI

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

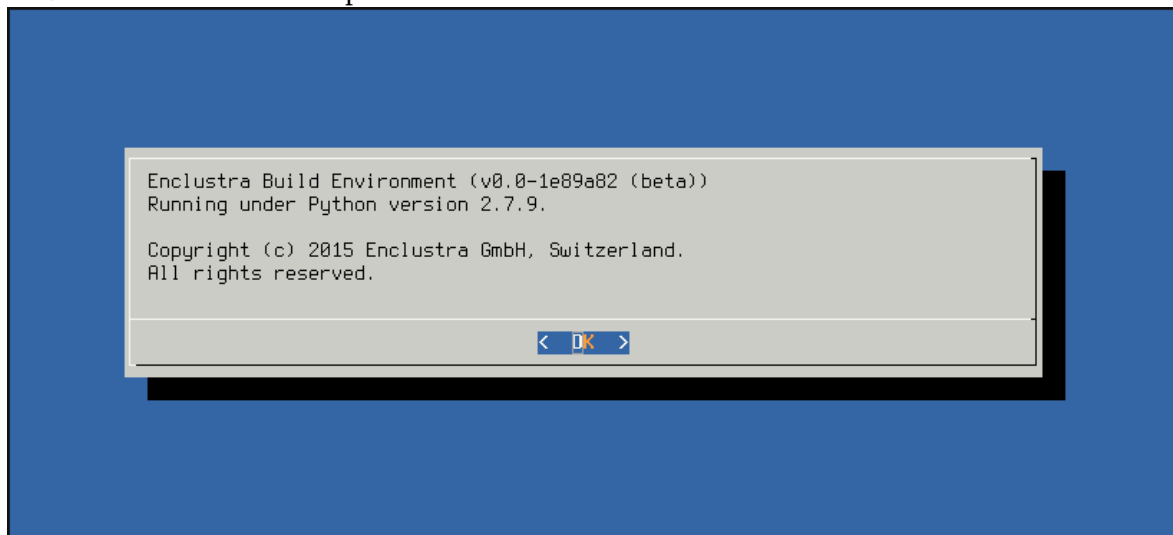1. Clone the build environment repository with:

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

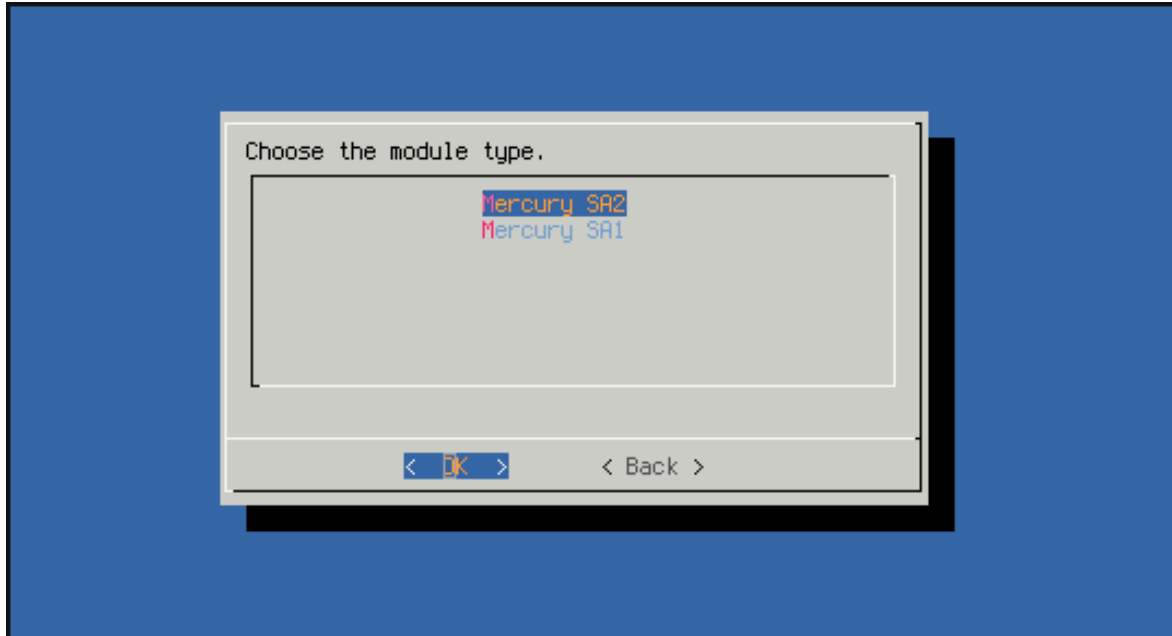2. Change to the enclustra-buildscripts directory:

```
cd enclustra-buildscripts
```
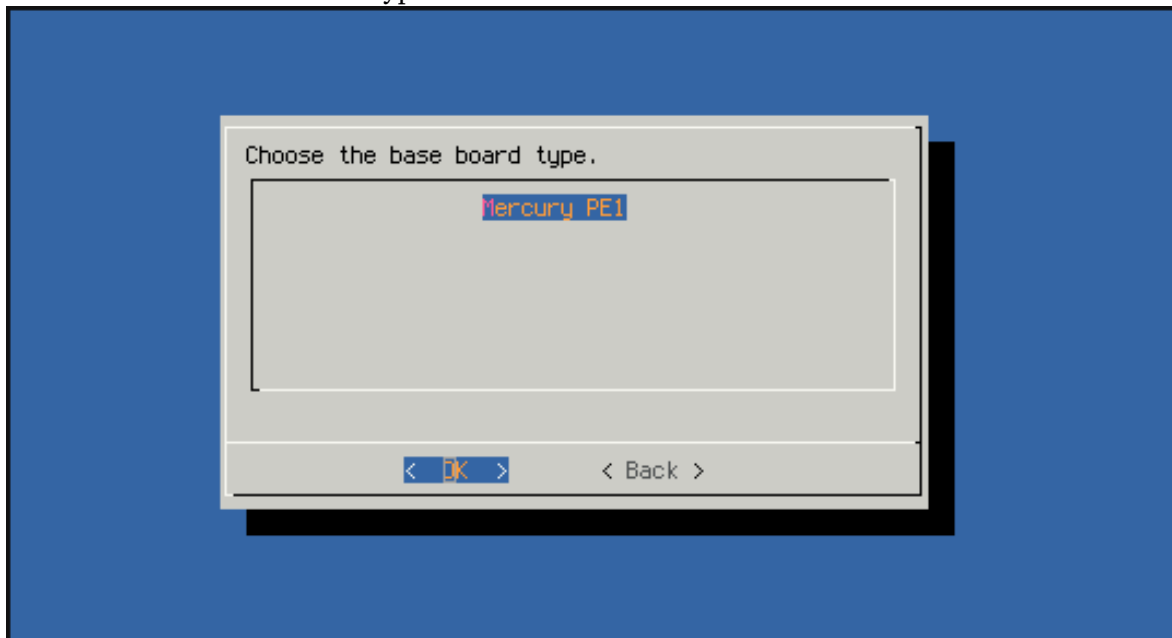
3. Run `./build.sh` script.

4. The welcome screen provides basic information about the version of the build environment.
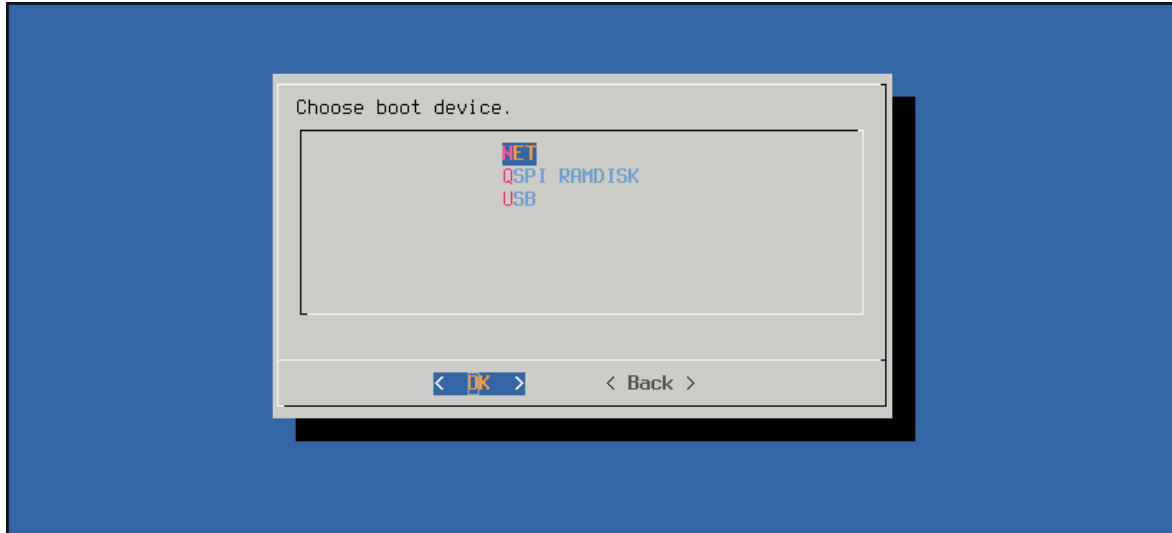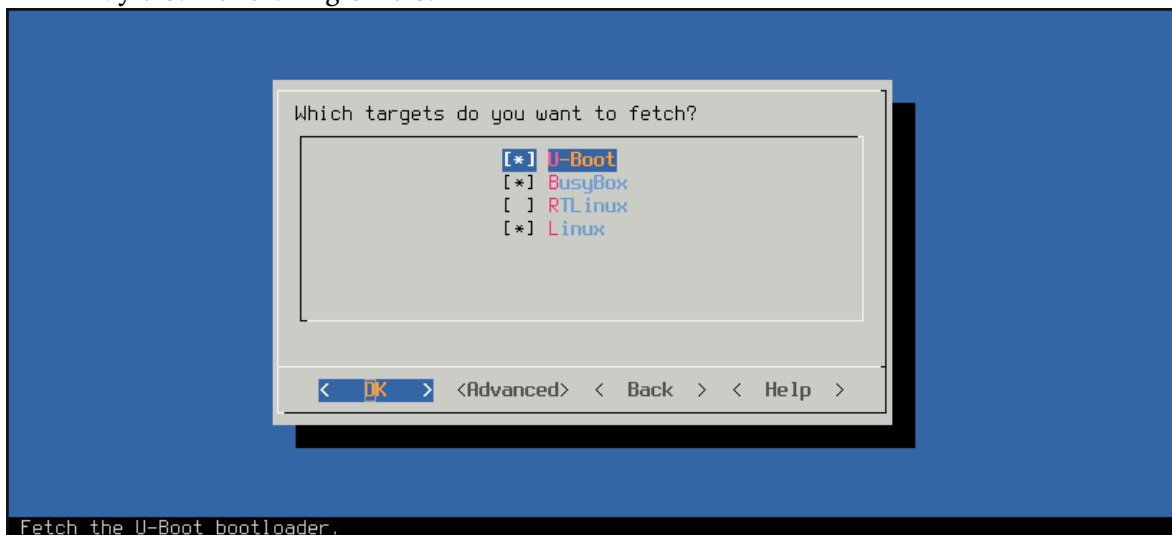

5. Choose the module type.

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.
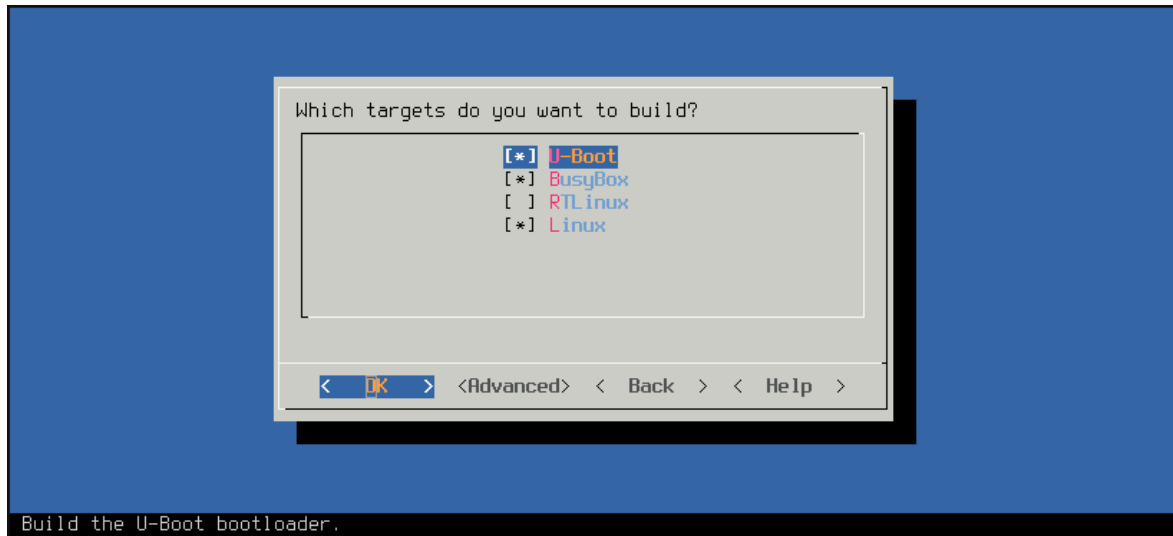
10. Choose the exact version of the device (chip type, industrial/commercial grade, speed grade).
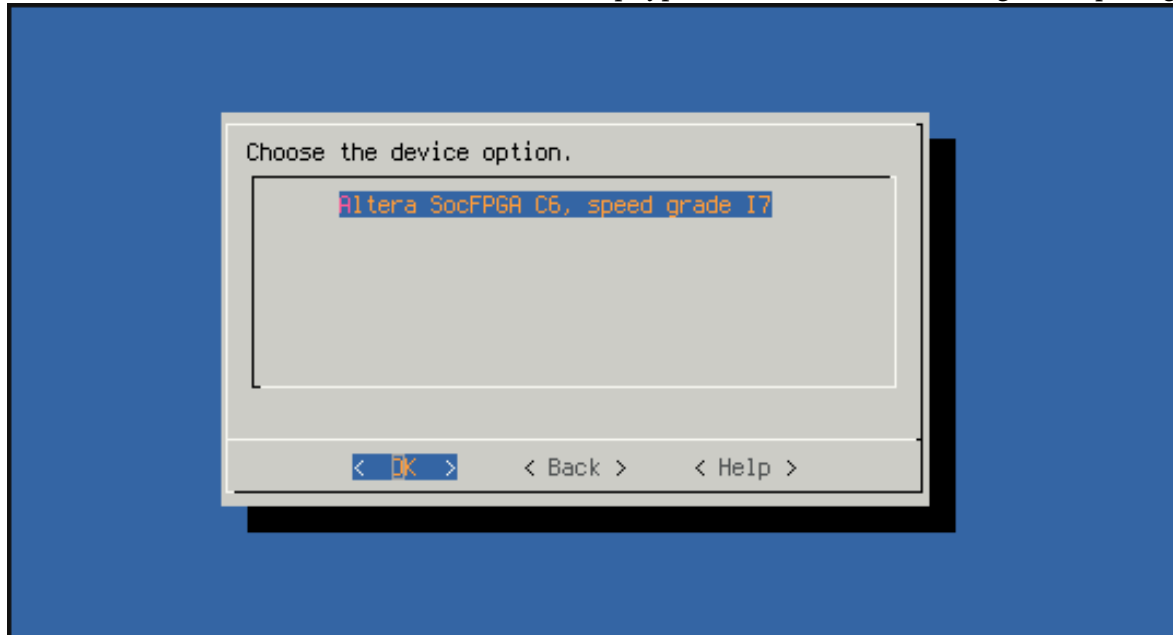
11. Verify all the chosen build parameters.
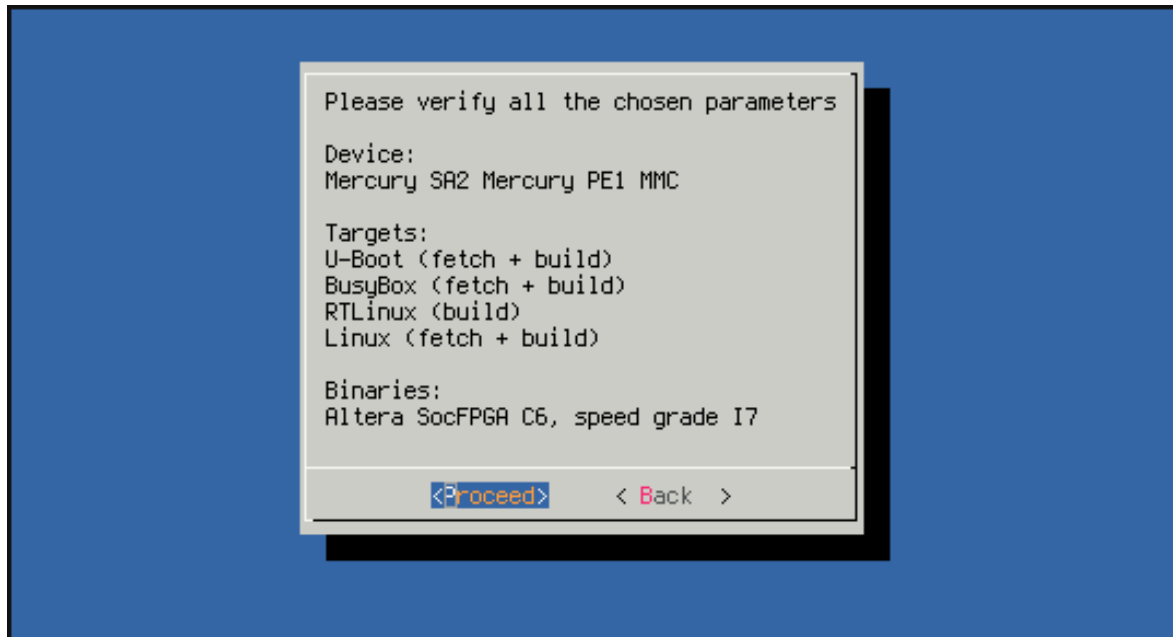
12. Choose wheter or not to save the configuration for later use.
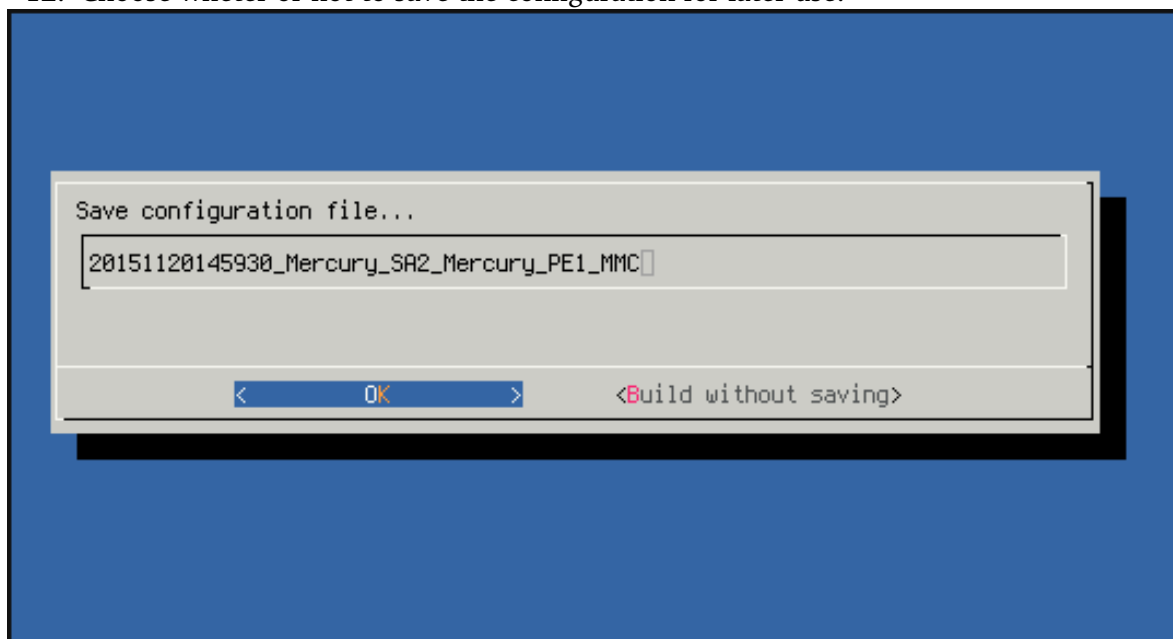
13. The build environment will fetch and build the chosen targets.

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 *Mercury SA1* module on the *Mercury PE1* base board in *QSPI* boot mode, the command would look like this:

```
./build.sh -d Mercury_SA1/Mercury_PE1/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 Mercury_SA1/Mercury_PE1/QSPI -l
```

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

```
./build.sh -d Mercury_SA1/Mercury_PE1/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 Mercury_SA1/Mercury_PE1/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 Mercury_SA1/Mercury_PE1/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 Mercury_SA1/Mercury_PE1/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
```

4.3 SD Card (MMC)

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

1. Prepare the SD Card (refer to the *SD Card (MMC) Partitioning Guide*).

2. Record the preloader image to the unformatted partition of a SD Card (type a2):

    - In Linux:

```
sudo dd if=preloader-mkpimage.bin of=/dev/sdX2
sync
# where X is the letter of the device
```

    - In Windows:

```
#. Run the ``Win32 Disk Imager`` as administrator.
#. Select the drive letter representing the SD Card in the device drop-down list.
#. Open the ``preloader-mkpimage.bin`` file.
#. Hit ``Write``.
#. Wait for completion and exit the application.
```

3. Mount the BOOT and rootfs partitions.

4. Copy `uImage`, `devicetree.dtb`, `fpga.rbf`, `u-boot.img` and `uboot.scr` from the build environment output directory to the BOOT partition (FAT formatted).

5. 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
```

6. Unmount all partitions mounted from the SD Card.

7. Insert the card into the SD Card slot on the board.

8. Configure the board to boot from the SD Card (refer to the board User Manual).

9. Power on the board.

10. 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 mmcboot
```

4.4 QSPI Flash

Table 4.1: Altera Family QSPI Flash Layout

Partition	Offset	Size
Preloader	0x0	0x40000
U-Boot image	0x60000	0x40000
FPGA Bitstream	0x100000	0x700000
U-Boot environment	0x800000	0x40000
Linux Device Tree	0x840000	0x40000
Bootscript	0x880000	0x40000
Linux kernel	0x8C0000	0x740000
JFFS2 Rootfs	0x1000000	0x3000000

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

1. Setup an TFTP server on the host computer.

2. Power on the board and boot to the U-Boot (e.g. from a MMC card).

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:

```
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 uImage, devicetree.dtb, uboot.scr, rootfs.jffs2, u-boot.img, fpga.rbf.img and preloader-mkpimage.bin from the build environment output directory to the TFTP server directory

7. Update the preloader image:

```
mw.b ${preloader_loadaddr} 0xFF ${preloader_size}
tftpboot ${preloader_loadaddr} ${preloader_image}
sf probe
sf erase ${qspi_preloader_offset} ${preloader_size}
sf write ${preloader_loadaddr} ${qspi_preloader_offset} ${filesize}
```

8. Update the U-Boot image:

```
mw.b ${uboot_loadaddr} 0xFF ${qspi_uboot_erase_size}
tftpboot ${uboot_loadaddr} ${uboot_image}
sf probe
sf erase ${qspi_uboot_erase_offset} ${qspi_uboot_erase_size}
sf write ${uboot_loadaddr} ${qspi_uboot_offset} ${filesize}
```

9. Update the bitstream image:

```
mw.b ${bitstream_loadaddr} 0xFF ${bitstream_size}
tftpboot ${bitstream_loadaddr} ${bitstream_image}
sf probe
sf erase ${qspi_bitstream_offset} ${bitstream_size}
sf write ${bitstream_loadaddr} ${qspi_bitstream_offset} ${filesize}
```

10. 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}
```

11. 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}
```

12. 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}
```

13. Update the rootfs image:

```
mw.b ${rootfs_loadaddr} 0xFF ${rootfs_size}
tftpboot ${rootfs_loadaddr} ${rootfs_image}
sf probe
sf erase ${qspi_rootfs_offset} ${rootfs_size}
sf write ${rootfs_loadaddr} ${qspi_rootfs_offset} ${filesize}
```

14. Configure the board to boot from the QSPI flash (refer to the board User Manual).

15. Reset the board.

16. The board should boot the Linux system.

If one wants to manually trigger booting from the QSPI flash - following command have to invoked from the U-Boot command line:

```
run qspiboot
```

Note: Note that the steps from 7 to 13 can be invoked independently.

4.5 USB Drive

The Altera family devices cannot boot directly from a USB device. The preloader and the U-Boot have to be started from a SD Card (MMC) or QSPI Flash. Please refer to *SD Card (MMC)* or *QSPI Flash* to boot the U-Boot from MMC or QSPI. When the 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.

2. Create a ext2 formatted partition as the second partition on the drive. This partition should have at least 16 MiB.

3. Copy `uImage`, `devicetree.dtb` and `uboot.scr` from the build environment output directory to the FAT formatted partition.

4. Unpack root file system onto the ext2 partition of the USB drive. This has to be done as root.

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

5. Insert the USB drive into the USB port of the board.

6. Configure the board to boot from the MMC card or QSPI flash (refer to the board User Manual).

7. Power on the board.

8. Stop the U-Boot autoboot.

9. Trigger USB boot with

```
run usbboot
```

4.6 NFS

The preloader and the U-Boot have to be started from a 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. Prepare the SD Card (refer to the *SD Card (MMC) Partitioning Guide*).

2. Record the preloader image to the unformatted partition of a SD Card (type a2):

```
            sudo dd if=preloader-mkpimage.bin of=/dev/sdX2
            sync
            # where X is the letter of the device
```

3. Copy `fpga.rbf`, `u-boot.img` and `uboot.scr` from the build environment output directory to the BOOT partition.

4. Copy `uImage`, `devicetree.dtb` and `uboot.scr` from the build environment output directory to the TFTP folder.

5. 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
```

6. Insert the card into the SD Card slot on the board.

7. Configure the board to boot from the SD Card (MMC).

8. Power on the board and stop the U-Boot autoboot.

9. 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
```

10. Trigger NFS boot with:

```
    run netboot
```

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

4.7 SD Card (MMC) Partitioning Guide

1. Insert the SD Card into the card reader of the host computer.

2. If the partitions were mounted - unmount them:

```
    sudo umount /path/to/mountpoint
```

3. Run `fdisk` tool:

```
    sudo fdisk /dev/sdX
    # where X is the letter of the SD card
```

4. Within `fdisk` run the following commands:

```
# delete any existing partition repeat until all are deleted
d
# create a new primary partition
n
# choose primary
p
# set number to '2'
2
# leave default start sector, and set the size to 2 MiB
+2M
# change the partition type
t
# choose the second partition (may be chosen automatically if only one exists)
2
# set type to Altera Boot Partition
a2
# create a new primary partition
n
# set as primary
p
# set number to '1'
1
# leave default start sector, and set the size to 16MiB
+16M
# change the partition type
t
# choose the first partition
1
# set type to fat16
4
# create the third partition
n
# set as primary
p
# set number to '3'
3
# leave default start and end sector
# write changes to the disk
w
# leave 'fdisk' (might be happening automatically)
q
```

5. Format newly created partitions:

```
sudo mkfs.fat -n BOOT /dev/sdX1
sudo mkfs.ext2 -L rootfs /dev/sdX3
# where X is the letter of the SD card
```

4.8 NFS Preparation Guide

For development, it can be very handy to mount the root filesystem via NFS (nfsroot).

To prepare the host machine several preparatory steps are required.

The following servers need to be installed on the host machine, and configured properly:

- NFS server (e.g. on Ubuntu `nfs-kernel-server`)
- TFTP server (e.g. on Ubuntu `tftpd`)
- DHCP server (e.g. on Ubuntu `isc-dhcp-server`)

For demonstration purpose, the TFTP folder on the host is `/tftpboot`. It can be configured like this:

```
$ sudo mkdir /tftpboot
$ sudo chown nobody /tftpboot
$ nano /etc/xinetd.d/tftp
$ cat /etc/xinetd.d/tftp
service tftp
{
protocol = udp
port = 69
socket_type = dgram
wait = yes
user = nobody
server = /usr/sbin/in.tftpd
server_args = /tftpboot
disable = no
}
```

For demonstration purpose, the NFS export folder is `/nfs_exp`. In can be configured like this:

```
$ sudo mkdir /nfs_exp
$ sudo chown root:root /nfs_exp
$ sudo chmod 777 /nfs_exp
$ sudo nano /etc/exports
$ cat /etc/exports
/nfs_exp 192.168.1.0/24(fsid=0,rw,no_subtree_check,no_root_squash)
$ sudo exportfs -a
```

Sometimes it is also necessary to restart the NFS server:

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

To configure the DHCP server do this:

```
$ cat /etc/dhcp/dhcpd.conf
default-lease-time 600;
max-lease-time 7200;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option domain-name-servers 192.168.1.1, 192.168.1.2;
subnet 192.168.1.0 netmask 255.255.255.0 {
```

```
    range 192.168.1.10 192.168.1.255;
}
$ sudo /etc/init.d/isc-dhcp-server restart
```

To select your ethernet interface edit /etc/default/isc-dhcp-server:

```
$ sudo sed -i -r 's/INTERFACES="(.+)"/INTERFACES="eth1"/g' /etc/default/isc-dhcp-server
sudo /etc/init.d/isc-dhcp-server restart
```

On the target in the Linux console, the NFS folder from the host should now accessible like this

```
# mkdir /nfs
# mount -t nfs -o port=2049,nolock,proto=tcp,ro,vers=3 <server_ip>:/nfs_exp /nfs
# cd /nfs
```

Note: Note that currently our Linux only supports NFS version 3, not 4. So the target folder needs to be specified, and the version is 3 (vers=3).

Note: Take care of the permissions on the server. They should match those used on the client.

After configuring your system, you can now deploy the new boot images to your TFTP folder, and extract the rootfs TAR archive to your NFS folder.

FAQ

5.1 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
```

5.2 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.

```sh
#!/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.