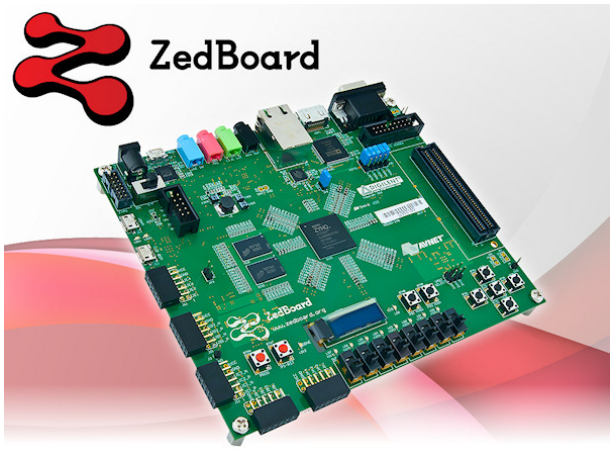
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Information Technology
and Electrical Engineering

ZedBoard at IIS

Guide



Pirmin Vogel
vogelpi@iis.ee.ethz.ch

Integrated Systems Laboratory

Version	Date	Editor	Comment
1.0	16.04.2014	vogelpi	Created
1.1	08.10.2014	vogelpi	Kernel module compilation added

Picture on title page: photograph of the Digilent ZedBoard [1].

Contents

List of Figures **III**

1 Introduction **1**

1.1 Outline 1

2 Overview **3**

3 Preparation **5**

4 Hardware Generation **7**

4.1 Exporting a Project from Vivado 7

4.2 Creating a Project in XPS 7

4.3 Device Tree Source File Generation 8

5 Software Generation **13**

5.1 Compilation of the U-Boot Boot Loader 13

5.2 Building the First Stage Boot Loader 14

5.3 Compilation of the Xilinx Linux Kernel 15

5.4 Generation of the Device Tree Blob 17

5.5 Root File System Generation using Buildroot 17

5.6 Customization of the Root File System 18

5.7 Wrapping the Image with a U-Boot Header 18

5.8 Preparation of the SD Card 19

5.9 Installation of the Linux System 19

5.10 Cross Compilation of Application Software 20

Bibliography **21**

A List of Abbreviations **23**

B Directory Trees **25**

B.1 Work Space 25

B.2 How-to Files 26

List of Figures

2.1 Overview of the system build process 4

4.1 Vivado export dialog window 7
4.2 XPS new BSB project wizard 8
4.3 SDK new hardware project wizard 9
4.4 SDK new BSP project wizard 9
4.5 SDK device tree BSP settings 10

5.1 SDK new application project wizard 14
5.2 SDK create zynq boot image dialog box 16
5.3 U-Boot welcome screen in minicom 20

Chapter 1

Introduction

Combining a powerful ARM Cortex-A9 dual-core processing system with a field-programmable gate array (FPGA) of the latest technology on a single chip, the Xilinx Zynq-7000 All-Programmable System on Chip (SoC) family [2] enables new levels of system integration and design flexibility. In the past 18 months, a variety of development kits featuring these attractive devices has been introduced. Examples include the ZC706 and ZC702 from Xilinx, the Parallela Board from Adapteva, which combines a Zynq SoC with an Epiphany multicore processor consisting of up to 16 reduced instruction set computing (RISC) cores [3], and the ZedBoard and the MicroZed from Digilent [4], which offer a reasonable trade-off between hardware capabilities and cost.

In our lab, there are several use cases for the Zynq platform. Examples include but are not limited to the development and testing of hardware accelerators and the corresponding software drivers, integration of individual hardware blocks into embedded systems, e.g., for rapid prototyping or demonstration purposes, and for building testbeds for wireless cellular communication. Irrespective of the application, the processing system of the Zynq platform requires an operating system to be of use. Most of the development kits come with complete images out of the box that allow Linux to be booted on the platform. However, most vendors have their own code sources and development environments, their own customizations and bugs, and also their own communities.

The goal of this how-to is to guide you through the process of building your own Linux system that

- is based on the Xilinx sources for the boot loader and the Linux kernel,
- works within the Integrated Systems Laboratory (IIS) infrastructure,
- you are under complete control of, and hence
- you can configure and customize to your needs.

This how-to is targeted at the generation of a Linux system for the ZedBoard. However, since the Xilinx sources are used, it can be used to generate a Linux system for any board based on the Xilinx Zynq-7000 All-Programmable SoC. Finally, you will have the operating system for your embedded system such that you can interface and test your hardware. Since this system is based on the Xilinx sources, you will be able to profit from a large community in case you run into troubles. Moreover, you won't have to trust configuration files or a kernel images that you find somewhere on the web. You will have a complete toolchain to generate everything you need yourself!

1.1 Outline

The remainder of this how-to is organized as follows. Chapter 2 gives an overview of the system build process. Some information about the development environment and preparation steps is given in Chapter 3. The main

topic of this guide, i.e., a step-by-step guidance to build an embedded system for the ZedBoard from the Xilinx sources is split in two parts. The build process of the hardware part is presented in Chapter 4. How to build the software part is then shown in Chapter 5.

Chapter 2

Overview

It makes sense to give the reader an overview of the system build process and how the different steps are related before providing a step-by-step guidance. The whole process is visualized in Figure 2.1.

First of all, the hardware part of the target system needs to be designed. This can be achieved by using the Xilinx Vivado Design Suite. Alternatively, you can use the Base System Builder (BSB) wizard in Xilinx Platform Studio (XPS). This results in the bitstream used to configure the Programmable Logic (PL), i.e., the FPGA part, of the Zync-7000 All Programmable SoC on the ZedBoard. The Xilinx Software Development Kit (SDK) is then used to create the Device Tree Source (DTS) file, which can be viewed as a summary of the hardware configuration. It provides information about the available hardware resources and their addresses to the Linux kernel that will later run on the ZedBoard. The device tree file is the only interface between the hardware and software build processes.

On the software side, the first task is to compile the U-Boot boot loader for the ZedBoard. To do so, a cross-compilation toolchain provided by Xilinx is used. The next step is then to build the First Stage Boot Loader (FSBL) which is the first piece of software that runs on the target system during startup. First, it configures the Processing System (PS) of the Zynq SoC, then loads the bitstream to the FPGA part, and finally starts the execution of U-Boot which then loads the Linux kernel.

The operating system itself consists of the separately generated Linux kernel, which establishes the software interface between the hardware and the higher-level application software, and the root file system that contains the actual programs. To build the file system, Buildroot and BusyBox are used. The file system obtained then needs to be uncompressed to add custom files such as scripts and previously generated Secure Shell (SSH) keys, and to edit others, e.g. the file system table to mount a network file system. Then, the edited file system needs to be compressed again.

Finally, the FSBL, the Device Tree Blob (DTB) obtained from the DTS file, the kernel and the root file system image are copied to a Secure Digital (SD) card from which the target system can be booted.

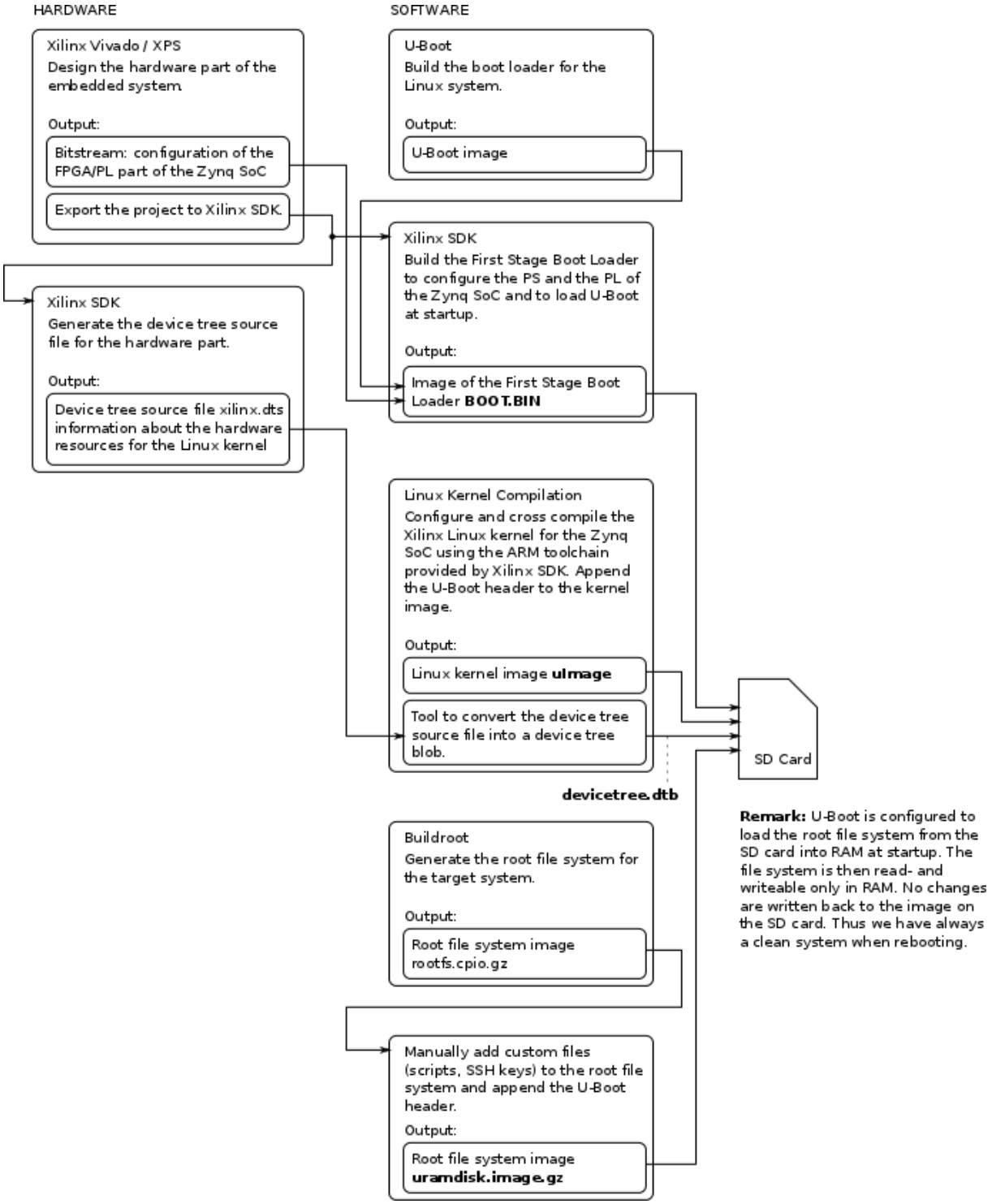


Figure 2.1: Overview of the system build process.

Chapter 3

Preparation

Before starting with the generation of the hard- and software for the ZedBoard, the required file system hierarchy needs to be established. As the the whole build process generates a lot of data that need to be stored and read from the hard disk, we recommend to place the working directory on the scratch drive of your local machine. As opposed to working on your home directory, this saves a lot of network traffic and speeds up the compilation of the source files. However, you should keep in mind that the local scratch drive is not backed up. Therefore you should save your configuration and output files to your home directory on a regular basis.

Throughout this how-to, we assume our workspace directory to be

```
/scratch/username/zedboard/workspace/ .
```

Copy the content of the folder `files/sample_workspace/` to your workspace to establish the proper directory structure described in Appendix B.1. The scripts and configuration files used in this how-to are provided in the folder `files/how-to_files/`, see Appendix B.2 for an overview of the provided files. Furthermore, we used Xilinx Vivado Design Suite 2014.1 installed on Software Installation and Sharing System (SEPP).

Chapter 4

Hardware Generation

In general, we use the Xilinx Vivado Design Suite 2014.1 to create the hardware part of our systems. However, this task is not a piece of cake and is worth an extended tutorial itself. Therefore, it is not part of this guide. Instead, we will show in Section 4.1 how to export the required files from Xilinx Vivado such that the software part of the embedded system can be generated. Alternatively, the BSB in Xilinx Embedded Development Kit (EDK) 14.7 can be used to generate a very basic Zynq design. This process is shown in Section 4.2.

4.1 Exporting a Project from Vivado

Open a terminal, navigate to the working directory and type

```
vivado-2014.1 vivado
```

to start Xilinx Vivado Design Suite. Open the project you want to generate the software part for. In the Flow Navigator, open the IP Integrator and click Open Block Design. Then, select Generate Block Design and hit Generate. To export the system to Xilinx SDK, select File → Export → Export Hardware for SDK... . A dialog window similar to the one in Figure 4.1 should open. Enter the paths as shown in the figure and hit the OK button.

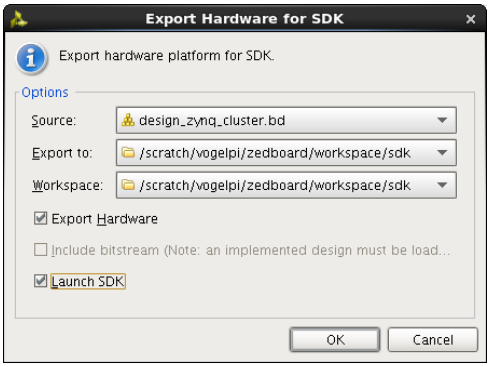


Figure 4.1: Vivado export dialog window.

4.2 Creating a Project in XPS

A base Zynq hardware design can be created using BSB in XPS. To do so, start Xilinx XPS by typing

```
xilinx-14.7 xps
```

and click File → New BSB Project. Enter the path to store the project file as indicated in Figure 4.2. The

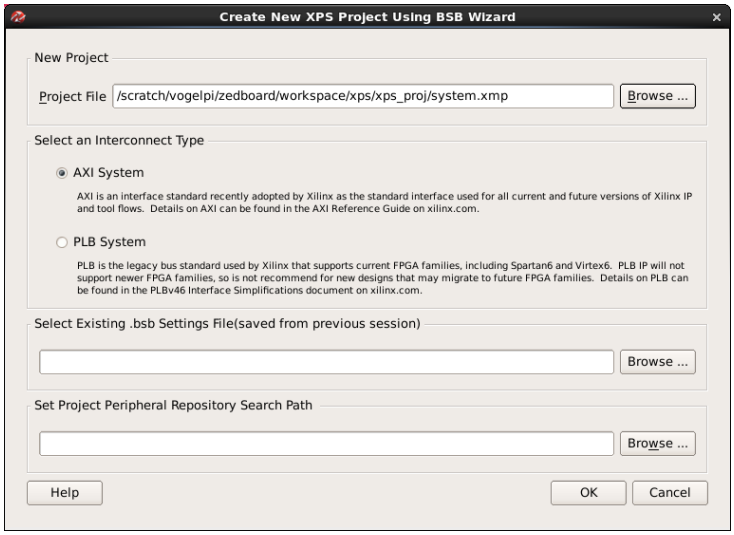


Figure 4.2: XPS new BSB project wizard.

peripheral configuration window opens. Leave everything as it is and click Next. XPS is now setting up the project.

After the project has been set up, custom hardware intellectual property cores (IPs) could be added to the design. For this version of the tutorial, we don't do that and just continue with the generation of the bitstream configuration file for the FPGA. To do so, click Hardware → Generate Bitstream.

Finally, click Project → Export Hardware Design to SDK.

4.3 Device Tree Source File Generation

The device tree can be viewed as a summary of the hardware configuration of the system and is the only interface between the hardware and software build processes. It contains information about the hardware resources and their addresses and is required for the Linux kernel to run on the ZedBoard. To generate the device tree, the Xilinx Device Tree Generator can be used. Here, we show how to generate the device tree file for our system using the Xilinx Device Tree Generator and Xilinx SDK. For more information about the Xilinx Device Tree Generator and the device tree in general, we refer to the online guide from Xilinx [5] and the Digilent Embedded Linux Development Guide [6], respectively.

Open a terminal, navigate to `/scratch/username/zedboard/workspace/device_tree_generator` and type

```
git clone git://git.xilinx.com/device-tree.git \
bsp/device-tree_v0_00_x
```

to download the Xilinx Device Tree Generator.

Then, go back to Xilinx SDK or start it by typing

```
vivado-2014.1 xsdk
```

and select `/scratch/username/zedboard/workspace/sdk` as workspace directory. Open the project exported from Vivado in Section 4.1. Alternatively, create a new project under File → New → Project. Select

Xilinx → Hardware Platform Specification and click Next. Enter the project name and specify the path to the system.xml file as shown in Figure 4.3. Click Finish.

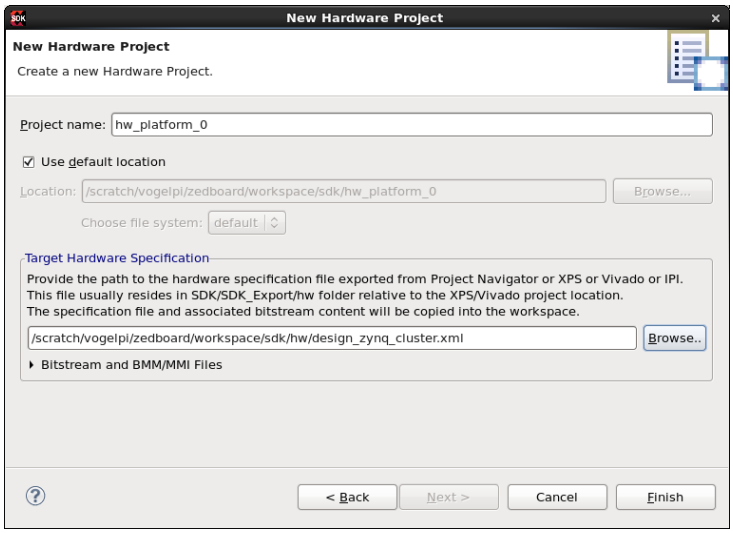


Figure 4.3: SDK new hardware project wizard.

Next, select Xilinx Tools → Repositories. Add a new local repository by clicking on New, enter the path to the directory containing the device tree generator, i.e., the bsp directory downloaded earlier:

```
/scratch/username/zedboard/workspace/device_tree_generator
```

and click OK. Close the Preferences window by clicking OK.

Now, the device tree for our project can be generated. To do so, generate a new Board Support Package (BSP) project by clicking File → New → Xilinx Board Support Package. A dialog box similar to the one shown in Figure 4.4 will be displayed. Select device-tree as Board Support Package OS and click Finish.

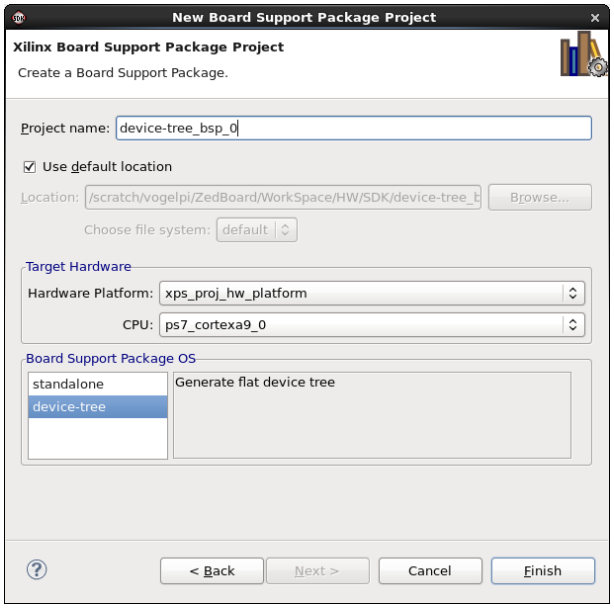


Figure 4.4: SDK new BSP project wizard.

In the next dialog, the boot arguments for the Linux kernel and the console device to use can be specified. Later, this dialog box can be opened by right-clicking on the device-tree_bsp_0 project in SDK's project explorer and then selecting Board Support Package Settings.

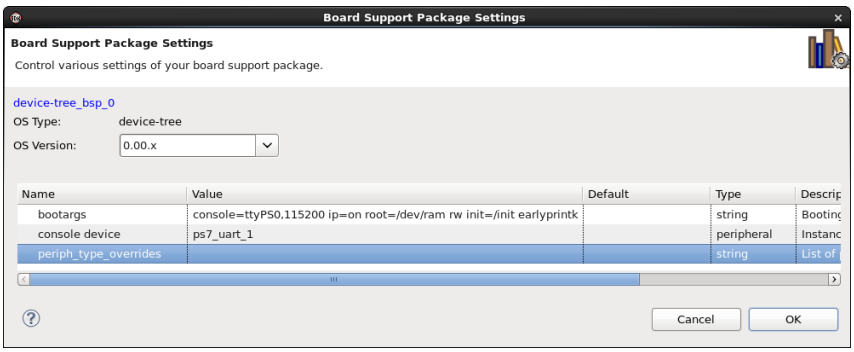


Figure 4.5: SDK device tree BSP settings.

Enter the values as shown in Figure 4.5 and click OK. By default, the device tree generator immediately runs to create the DTS file. If not, right-click on the device-tree_bsp_0 project in the project explorer and select Build Project.

Entering

```
console=ttyPS0,115200 ip=on root=/dev/ram rw init=/init earlyprintk
```

as boot arguments as shown in Figure 4.5, the kernel will

- use the ttyPS0 device as default serial console with a baudrate equal 115200 baud,
- mount a random access memory (RAM) disk image as read and writeable root file system[1],
- search for and execute the init script in `/init` and
- show early boot messages on the default serial console.

With the `ip=on` statement in the boot arguments, Dynamic Host Configuration Protocol (DHCP) support for the system is activated. When connected to a network, it tries to get an Internet Protocol (IP) address from the DHCP server. In the iis-spec network at the IIS, the DHCP server looks up the Media Access Control (MAC) address of the client and assigns a fixed IP address specified in a table to it. At the time of writing, there are three ZedBoards in the Digital Circuits and Systems research group at IIS and three IP addresses reserved in the iis-spec network. The corresponding MAC addresses are shown in Table 4.1. Actually, it should be possible to pass the MAC address to the kernel via DTS file but for some reason, this seems not to be working with the current version of the tools. As a consequence, we will show in Section 5.9 how to set the MAC address as an environment variable in the boot loader configuration menu. Note that access to the ZedBoards over Ethernet is only possible from within the IIS network. We have used ZedBoard 1 for this how-to.

ZedBoard	MAC Address	IP Address
1	00 0A 35 00 01 A1	172.31.28.180
2	00 0A 35 00 01 A2	172.31.28.181
3	00 0A 35 00 01 A3	172.31.28.182

Table 4.1: Reserved MAC and IP addresses for the ZedBoards in the iis-spec network.

To finish this section, two adjustments to the generated DTS file `xilinx.dts` need to be done. To do so, open the DTS file in

[1]Note that the address at which the RAM disk image is found in RAM is passed to the kernel via boot loader

```
/scratch/username/zedboard/workspace/sdk/device-tree_bsp_0/\
ps7_cortexa9_0/libsrc/device-tree_v0_00_x/
```

and search for the device node ps7_ethernet_0.

1. Search for the property local-mac-address and delete it. We do not need to set the MAC address in the device tree since we intend to use the boot loader to pass it to the kernel.
2. Search for the child node phy0 of mdio and change its properties from

   ```
   compatible = "marvell,88e1510";
   device_type = "ethernet-phy";
   reg = <7>;
   ```

   to

   ```
   compatible = "marvell,88e1116r";
   device_type = "ethernet-phy";
   reg = <0>;
   ```

   to make sure that the kernel will load the proper driver and use the proper physical address of the device register for the Ethernet device.

Save and close the DTS file.

Chapter 5

Software Generation

Xilinx Vivado 2014.1 provides various toolchains that allow to cross compile software for many different systems generated using the toolsuite including the PS on the ZedBoard. Consequently, we can directly start to build the boot loader and the Linux kernel for the hardware system generated in Chapter 4.

5.1 Compilation of the U-Boot Boot Loader

We use the universal boot loader U-Boot to load the images of the Linux kernel and the root file system, and providing the device tree to the kernel during the boot process of the system. It supports many different architectures, e.g., PowerPC, ARM, x86 and target boards such as the ZedBoard. For more information about how to use U-Boot on Xilinx boards, we refer to the corresponding entry in the Xilinx online wiki [7].

Open a terminal and navigate to the workspace directory. Then, type

```
git clone git://github.com/Xilinx/u-boot-xlnx.git
```

to download and unpack the U-Boot source tree. Before building U-Boot, we will now adjust its configuration. As mentioned in Section 4.3, we use the boot loader to pass the Ethernet MAC address to the kernel. However, instead of setting a fixed MAC address in the configuration of U-Boot, which would require U-Boot as well as the FSBL to be separately rebuilt for every board, we will now configure U-Boot such that the MAC address can be set using the serial console. To do so, open the configuration file zynq-common.h in `u-boot-xlnx/include/configs`. Search and delete or comment the lines

```
#define CONFIG_IPADDR    10.10.70.102
#define CONFIG_SERVERIP  10.10.70.101
```

to prevent U-Boot from obtaining a fixed IP address. In case that the system is to be booted over the network using Trivial File Transfer Protocol (TFTP), the second line is used to tell U-Boot the IP of the server providing the kernel image. Just like the MAC address, these two parameters could also be set later using the serial console if required. To finish the configuration, search and delete the assignment

```
"ethaddr=00:0a:35:00:01:22\0" \
```

and add the line

```
#define CONFIG_ENV_OVERWRITE 1
```

at the end of the file (above the final `#endif` statement) to allow the environment variables to be changed later using the serial console.

Now, the U-Boot can be built. Copy the compilation script `compile_loader.sh` to the U-Boot folder. This simple script sets the `CROSS_COMPILE` variable to point to the proper cross-compilation toolchain provided by Xilinx SDK, starts the compilation of U-Boot, and copies the generated Executable and Linkable Format (ELF) image to the folder `boot_image` in the workspace directory. To start the compilation of U-Boot using, e.g., four processor cores, simply type

```
vivado-2014.1 ./compile_loader.sh -j4
```

into your terminal.

5.2 Building the First Stage Boot Loader

Now that we have the bitstream configuration file for the FPGA and the bootloader for the Linux kernel, the FSBL can be generated. Once loaded, this piece of code configures the FPGA with the bitstream before loading U-Boot.

Open a terminal and navigate to `/scratch/username/zedboard/workspace/sdk`. Open Xilinx SDK by typing

```
vivado-2014.1 xsdk
```

and select `/scratch/username/zedboard/workspace/sdk` as workspace directory.

Click File → New → Project. Select Xilinx → Application Project and click Next. A dialog box similar to the one shown in Figure 5.1 will be displayed. Enter the values specified in the figure and hit Next. In the template dialog box, select Zynq FSBL and click Finish.

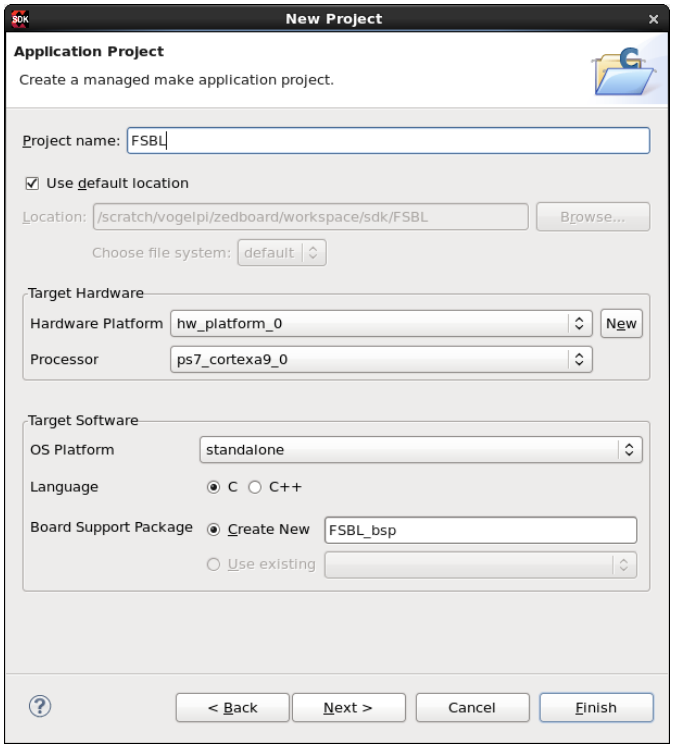


Figure 5.1: SDK new application project wizard.

On the ZedBoard, the Universal Serial Bus (USB) physical layer (PHY) chip needs to reset by the FSBL.

To do so, go to the Project Explorer → FSBL and open the file `main.c` in the folder `src`. Search for `FsblMeasurePerfTime(tCur,tEnd);` and add the following segment of code below the `\#endif` statement.

```
1   /* Reset the USB */
2   {
3     fsbl_printf(DEBUG_GENERAL, "Reset USB...\r\n");
4
5     /* Set data dir */
6     *(unsigned int *)0xe000a284 = 0x00000001;
7
8     /* Set OEN */
9     *(unsigned int *)0xe000a288 = 0x00000001;
10    Xil_DCacheFlush();
11    /* For REVB Set data value low for reset, then back high */
12  #ifdef ZED_REV_A
13    *(unsigned int *)0xe000a048 = 0x00000001;
14    Xil_DCacheFlush();
15    *(unsigned int *)0xe000a048 = 0x00000000;
16    Xil_DCacheFlush();
17  #else
18    *(unsigned int *)0xe000a048 = 0x00000000;
19    Xil_DCacheFlush();
20    *(unsigned int *)0xe000a048 = 0x00000001;
21    Xil_DCacheFlush();
22  #endif
23  }
```

Save your changes and check whether the project is rebuilding itself automatically. If it doesn't, click Project → Clean and then Project → Build All. The generated ELF file of the FSBL can be found in

```
/scratch/username/zedboard/workspace/sdk/FSBL/Debug/FSBL.elf
```

Now, the the binary boot image `BOOT.BIN` can be generated. To do so, click Xilinx Tools → Create Zynq Boot Image. A dialog window similar to the one shown in Figure 5.2 opens. Enter the files **in the same order** as shown in the figure and click Create Image. Note that in the newer versions of SDK, for each file, a file type must be specified. For `FSBL`, select bootloader and for the others, select datafile. The binary file is then created and named `u-boot.bin`. For the ZedBoard to find and load it, rename it to `BOOT.BIN`.

5.3 Compilation of the Xilinx Linux Kernel

Open a terminal and navigate to the workspace directory. Then, type

```
git clone git://github.com/Xilinx/linux-xlnx.git
```

to download and unpack the Xilinx Linux source tree. Enter the directory `linux-xlnx` and type

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
xilinx_zynq_defconfig
```

to load the default kernel configuration for systems using Xilinx ZYNQ. Now, it is time to customize the kernel if necessary. Type

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
menuconfig
```

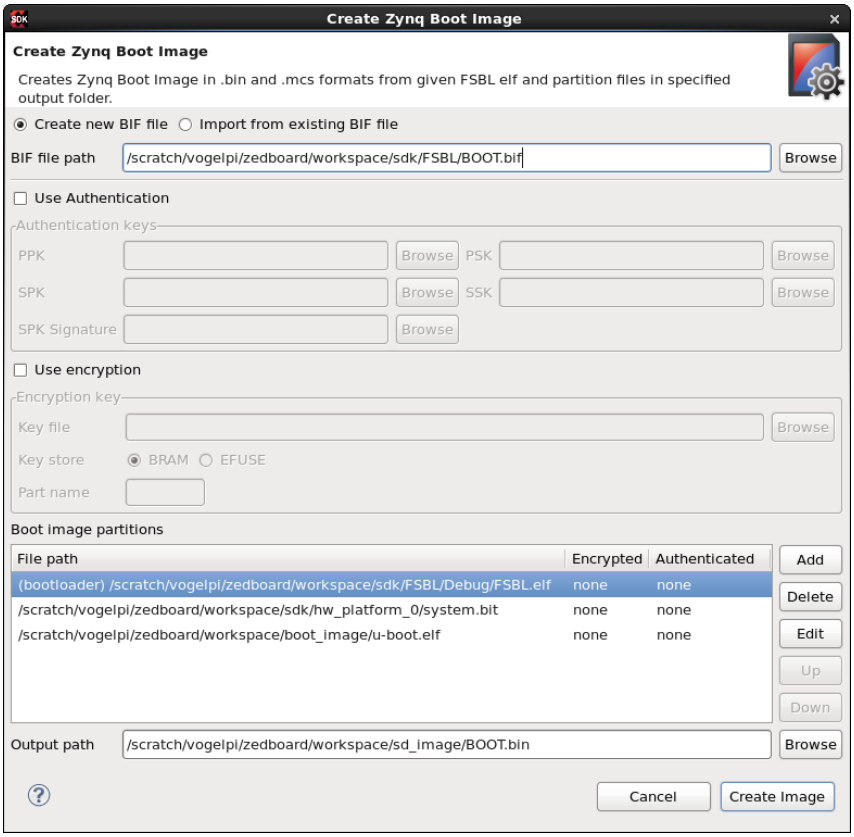


Figure 5.2: SDK create zynq boot image dialog box. It is important, that the files are entered in the same order for the FSBL to work properly.

into the terminal to load the configuration menu. It is recommended to activate the option File Systems → FUSE (Filesystem in Userspace) support in order to allow, for example, the use of SSH Filesystem (SSHFS) to mount directories located on your workstation, which can simplify software development for the target system quite a bit. Make your changes and save the configuration to `.config`.

To compile the kernel, copy the compilation script `compile_kernel.sh` to the directory. This script adds the U-Boot tools folder to the current `PATH` variable, sets the `CROSS_COMPILATION` variable, starts the compilation of the Xilinx Linux kernel and copies the generated kernel image to the folder `sd_image` in the workspace directory. The U-Boot tools, more precisely mkimage, are used to append an additional header to the kernel image. This header allows U-Boot to check that the kernel image has been properly loaded before starting the kernel. To start the kernel compilation using, e.g., four processor cores, simply type

```
vivado-2014.1 ./compile_kernel.sh -j4
```

into your terminal.

Modules that have been selected with `Y` during kernel configuration are already compiled during kernel compilation and included the generated kernel image. To support the loading and unloading of modules that have been selected with `M` at runtime using tools like `modprobe`, `insmod` and `rmmod`, the corresponding modules need to be compiled separately and added to the root file system in `/lib/modules/`, which can be achieved using the script `compile_modules.sh`. Copy the script to the linux kernel directory and type

```
vivado-2014.1 ./compile_modules.sh
```

to execute it. The generated file hierarchy including the compiled modules can then be found in `lib_modules/` and is also copied to the folder `root_file_system/custom_files` in the workspace directory.

5.4 Generation of the Device Tree Blob

To successfully boot the Linux kernel built in Section 5.3, the DTS file generated in Section 4.3 needs to be converted to a DTB which can be fed to the kernel. Note that the executable required for this conversion is only generated during the kernel compilation process.

Copy the script `generate_dtb.sh` to the folder `linux-xlnx` in the workspace directory. This script fetches the DTS file from the corresponding SDK project directory, compiles the DTB and saves it to the `sd_image` directory. It can be executed by simply typing

```
./generate_dtb.sh
```

into the terminal.

5.5 Root File System Generation using Buildroot

Next, we are going to generate the root file system for our Linux system using Buildroot. Buildroot is a set of makefiles and patches that simplifies and automates the generation of a complete embedded Linux system, including cross-compilation toolchain, root file system, kernel and boot loader image.

Open a terminal and navigate to the workspace directory. Then, type

```
git clone git://git.buildroot.net/buildroot.git
```

to download and unpack the buildroot source tree. Enter the directory `buildroot` and copy the Buildroot configuration file `buildroot-config` to this directory and rename it to `.config`.

Start the configuration menu using

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
menuconfig
```

and make sure that under Toolchain, the field Toolchain path matches the actual location of the toolchain, i.e.,

```
/usr/pack/vivado-2014.1-kgf/SDK/2014.1/gnu/arm/lin
```

and that the field Toolchain prefix matches the actual prefix, i.e.,

```
arm-xilinx-linux-gnueabi
```

Now, you can make your own adjustments to the root file system. You can for example add the support for other file system types, include different editors and so on.

An additional source for packages for the target system is BusyBox. It combines tiny versions of many common UNIX utilities into a single small executable and provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, and so on. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts.

In the directory `buildroot`, execute the command

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
busybox-source
```

to download the BusyBox sources. Then, copy the BusyBox configuration file `busybox-config` to the Buildroot directory. You don't need to rename it. Next, you can execute the command

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- \
busybox-menuconfig
```

to make adjustments to the BusyBox configuration.

Copy the compile script `generate_fs.sh` to the Buildroot directory and execute it to start the generation of the root file system by typing

```
vivado-2014.1 ./generate_fs.sh -j4
```

to your terminal. The first execution of this command will take quite some time. The script will copy the generated root file system to

```
root_file_system/rootfs.cpio.gz
```

in the working directory.

Note that whenever you add additional packages to an existing root file system, the whole root file system will grow in size. However, if you remove packages, it does not necessarily shrink. We therefore recommend to archive the configuration files together with the generated root file system before making any major changes in the configuration. When removing packages to reduce the size of the file system, we recommend to rebuild it from scratch. To do so, execute `make clean` before generating the root file system.

5.6 Customization of the Root File System

Note: For this step, you will need root privileges on your machine.

Next, the root file system for our Linux system generated in Section 5.5 can be customized. This step is of particular importance as long as any changes to the root file system are not persistent. By default, the root file system is copied into RAM at boot time, and is then read- and writeable only in RAM. No changes are written to the image on the SD card. Thus, the system is always in a clean state after a reboot.

As an example, we will now place a script inside `/etc/init.d/` that sets a root password to adf12?34 during boot up such that the system can afterwards be accessed through SSH. Moreover, we will store previously generated SSH keys under `/etc/` and `/etc/dropbear/`. To generate new SSH keys for your system, generate the system without SSH keys. At startup, they will be automatically generated. Transfer them to your computer via scp to include them in the file system.

Copy the script `customize_fs.sh` to the folder `root_file_system`. This script will make a copy of `rootfs.cpio.gz`, unpack it to the `tmp_mnt` folder, change the ownership of the files in `custom_files`, copy them to the root file system, and repack it.

5.7 Wrapping the Image with a U-Boot Header

Navigate to the directory `root_file_system` in the working directory. Copy the script `add_header.sh` to the current folder and execute it.

This script uses the mkimage tool from the folder `u-boot-xlnx` to append the proper U-Boot header to root file system image. The output `uramdisk.image.gz` is copied to the folder `sd_image`.

5.8 Preparation of the SD Card

Note: For this step, you will need root privileges on your machine.

To boot Linux on the ZedBoard from an SD card, we first need to assure the the card is set up correctly. In the following, we quickly show how to format the card and create the required partitions. For more information, refer to [8].

Connect the SD card to your computer and open a terminal. Type `df` to see whether any partitions on the card are currently mounted, and to get the current device node of the card, e.g., `/dev/sdd`. Unmount them by typing

```
sudo umount /path_to_mount_point
```

into your terminal. Then type

```
sudo fdisk /device_node_of_card
```

to start the format tool. Type `p` to list the current partition table. Delete any existing partitions by typing `d`.

Next, type `n` followed by `p` and `1` to create a new primary partition. Type `1` followed by `1G` to define the first and the last cylinder, respectively. Then, type `n` followed by `p` and `2` to create a second primary partition. Select the fist and last cylinder of this partition to fill the rest of the card. Type `p` to list the newly created partitions and to get their device nodes, e.g., `/dev/sdd1`. To write the partition table to the card and exit the tool, type `w`.

To create a file system in the new partitions, we use mkfs. Type

```
sudo mkfs -t vfat -n ZED_BOOT /device_node_of_1st_partition
```

to format the first partition to FAT and label it `ZED_BOOT`. Similarly, type

```
sudo mkfs -t ext4 -n -L ROOT_FS /device_node_of_2nd_partition
```

to format the second partition to ext4.

5.9 Installation of the Linux System

Connect the SD card prepared in Section 5.8 to your computer and copy the content of the folder `sd_image` to the first partition labeled `ZED_BOOT`. The following files should now be placed on the SD card:

- `BOOT.BIN`: The FSBL generated in Section 5.2
- `devicetree.dtb`: The DTB prepared in Sections 4.3 and 5.4
- `uImage`: The Linux kernel image compiled in Section 5.3
- `uramdisk.image.gz`: The root file system generated in Section 5.5, customized in Section 5.6, and finalized in Section 5.7.

Eject the SD card, and put it into the SD card slot of the ZedBoard. Next, check the jumper settings (MIO[5] = 1, MIO[4] = 1, MIO[3] = 0). Connect your computer to the USB-to-UART bridge of the ZedBoard and open a terminal.

After activating the power switch of the ZedBoard, type

```
minicom ttyACM0
```

to start minicom and connect it to the device node `/dev/ttyACM0`. You should now see the U-Boot welcome screen displayed over the ZedBoard's serial console output as shown in Figure 5.3. Press any button to interrupt the boot process. Enter the two commands

```
set ethaddr 00:0A:35:00:01:A1
saveenv
```

to manually set the MAC address of the Ethernet port according to Table 4.1. After saving the environment variables, the MAC address is permanently set. You can now enter `sdboot` to continue the boot procedure or reset the board to make it boot Linux.

Note that after executing `saveenv`, U-Boot will always load its environment variables from the flash memory on the board, which does not change even when you copy a new version to the SD card. To load the environment variables provided in the binary image on the SD card, type `env default -a`.

At the end of the boot procedure, you should see the login prompt of the Linux system. Congratulations! You have generated your embedded Linux system from scratch and just booted it!

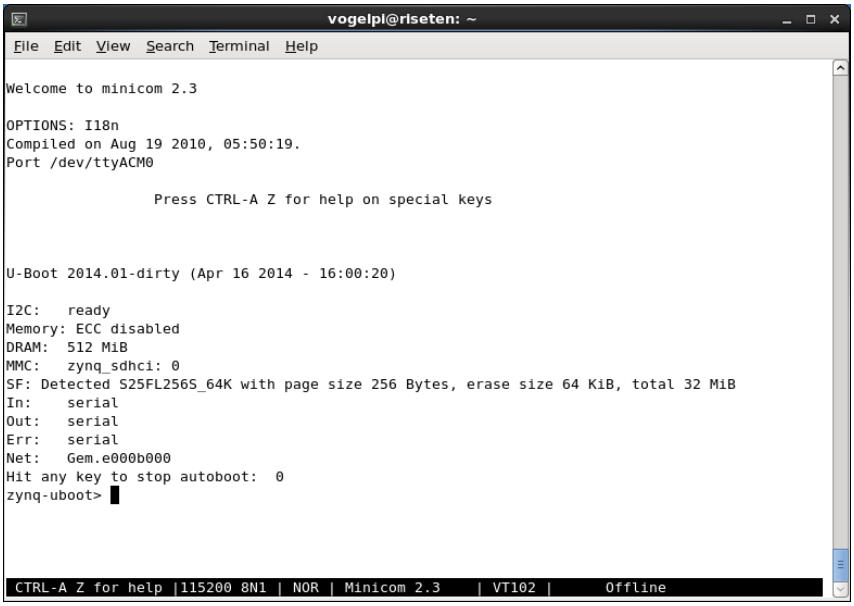


Figure 5.3: U-Boot welcome screen in minicom.

5.10 Cross Compilation of Application Software

To successfully compile your own software for the embedded Linux system running on the ZedBoard, you just need to call the proper compiler provided by Xilinx.

This can be achieved by simply calling your makefile like

```
vivado-2014.1 make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

as it has been done throughout this guide. An example makefile is shown below.

```
1 all: hello.c
2 	${CROSS_COMPILE}gcc -Wall -O0 hello.c -o hello
```

Bibliography

[1] "Digilent ZedBoard photograph," photograph, Digilent, Jul. 2013. [Online]. Available: http://www.digilentinc.com/Data/Products/ZEDBOARD/ZedBoard-obl-bg-600.jpg

[2] "Zynq-7000 All Programmable SoC," product page, Xilinx, Apr. 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

[3] "The Paralella Board," product page, Adapteva, Apr. 2014. [Online]. Available: http://www.adapteva.com/parallela-board/

[4] "ZedBoard.org," community-based web site, ZedBoard.org, Apr. 2014. [Online]. Available: http://www.zedboard.org

[5] "Build Device Tree Blob," online guide, Xilinx, Jul. 2013. [Online]. Available: http://www.wiki.xilinx.com/Build+Device+Tree+Blob

[6] "Digilent Embedded Linux Development Guide," guide, Digilent, Jan. 2013. [Online]. Available: http://www.digilentinc.com/Data/Products/EMBEDDED-LINUX/Digilent_Embedded_Linux_Guide.pdf

[7] "U-Boot," online wiki, Xilinx, May 2013. [Online]. Available: http://www.wiki.xilinx.com/U-boot

[8] "Digilent Getting Started With Embedded Linux," guide, Digilent, Jan. 2013. [Online]. Available: http://www.digilentinc.com/Data/Products/EMBEDDED-LINUX/ZedBoard_GSwEL_Guide.pdf

Appendix A

List of Abbreviations

BSB Base System Builder

BSP Board Support Package

DHCP Dynamic Host Configuration Protocol

DTB Device Tree Blob

DTS Device Tree Source

EDK Embedded Development Kit

ELF Executable and Linkable Format

FPGA field-programmable gate array

FSBL First Stage Boot Loader

IIS Integrated Systems Laboratory

IP intellectual property core

IP Internet Protocol

MAC Media Access Control

PHY physical layer

PL Programmable Logic

PS Processing System

RAM random access memory

RISC reduced instruction set computing

SDK Software Development Kit

SD Secure Digital

SEPP Software Installation and Sharing System

SoC System on Chip

SSH Secure Shell

SSHFS SSH Filesystem

TFTP Trivial File Transfer Protocol

USB Universal Serial Bus

XPS Xilinx Platform Studio

Appendix B

Directory Trees

In this chapter, an overview of the file system hierarchy is given.

B.1 Work Space

Copy the content of `files/sample_workspace/` to your own directory on the scratch volume of your machine to establish the file system hierarchy presented below. The folders in italic font will be created during the tutorial, e.g., when checking out a git repository.

workspace
- boot_image — components of the FSBL
- *buildroot* — buildroot sources
- device_tree_generator — device tree generator source files
- *linux-xlnx* — Xilinx Linux kernel sources
- root_file_system — images of the root file system
  - custom_files — custom files to be included in the root file system image
  - tmp_mnt — folder to unpack the root file system image to for customization
- sdk — project and source files for Xilinx SDK
- sd_image — output files of the build process, to be copied to the SD card
- *u-boot-xlnx* — Xilinx U-Boot sources
- xps — project files for XPS

B.2 How-to Files

Under `files/how-to_files/` you can find the scripts, configuration files and code segments required throughout this tutorial. Below, an overview of these files is given. The folder in italic font will be generated during when checking out the corresponding git repositories. The files shown below need to be copied to these directories after checking out the respositories.

how-to_files
- *buildroot*
  - add_header.sh . . . . . script to add a U-Boot header to a root file system image
  - buildroot-config . . . . . configuration file for Buildroot, rename it to .config
  - busybox-config . . . . . configuration file for BusyBox
  - customize_fs.sh . . . . . script to add custom files to a root file system image, requires root privileges
  - generate_fs.sh . . . . . script to start the root file system generation using Buildroot
- *linux-xlnx*
  - compile_kernel.sh . . . . . compile script for the Xilinx Linux kernel, adds U-Boot header to the image
  - compile_modules.sh . . . . . compile script for the loadable kernel modules
  - generate_dtb.sh . . . . . script to generate the DTB file from the DTS file
  - kernel_config . . . . . example configuration file for the Linux kernel
- sdk
  - segment_for_main.c . . . . . code segment to reset the USB PHY on the ZedBoard by the FSBL
- *u-boot-xlnx*
  - compile_loader.sh . . . . . compile script for U-Boot
  - zynq-common.h . . . . . adjusted U-Boot configuration files for Zynq systems