Linux on the ZedBoard

Tong Wu

February 29, 2016

Contents

1 The Basics of Running Linux on the ZedBoard **2**
1.1 Introduction 2
1.2 Booting Linux on the ZedBoard 2
1.2.1 Preparing the ZedBoard and the SD-Card 3
1.2.2 Booting the Zynq Device 4
1.2.3 Connecting to the ZedBoard over SSH 4
1.3 Configuring the FPGA at Runtime under Linux 5
1.3.1 Creating a Simple GPIO Project Using Vivado 5
1.3.2 Configuring the FPGA 7
1.4 Interfacing with the FPGA using GPIO under Linux 7
1.4.1 Creating a C/C++ Project in Xilinx SDK 7

2 Building the Boot System from Scratch (Advanced) **9**
2.1 Introduction 9
2.2 Building the Device Tree Compiler 9
2.3 Building Das U-Boot 10
2.3.1 Configure for Booting Over the Network (Optional) 10
2.3.2 Building U-Boot 11
2.4 Building the Linux Kernel 11
2.5 Building the First Stage Boot Loader and Creating BOOT.bin 12
2.5.1 Building the FSBL 12
2.5.2 Creating BOOT.bin 12
2.6 Modifying the Root File System 13
2.7 Setting Up the TFTP Server (Optional) 13

Chapter 1

The Basics of Running Linux on the ZedBoard

1.1 Introduction

The aim of this guide is to get you up and running with Linux on the ZedBoard. We will see how to boot Linux on the ZedBoard so that we can access a bash shell over SSH, configure the FPGA at run-time under Linux and execute a program to interface with the FPGA and toggle some LEDs.

First let's see why we would want to run Linux on the ZedBoard;

- Linux provides a familiar programming model, such as thread management. Whereas when programming on baremetal, it is difficult to perform separate tasks on multiple threads, Linux provides this functionality for us in the form of processes.
- Easy access to the IP network using the built in Linux network stack.
- Interact with the device using SSH over the network. This is a much more stable alternative over serial communications, which is the standard way of communication when programming baremetal applications.
- File transfer using SCP, enables us to quickly transfer files over the network.
- Dynamic FPGA configuration, easily reconfigure the FPGA on the fly, no need for reboot.

For this guide, it is assumed that you have **Xilinx Vivado and SDK 2015.4** installed. Additionally, for Windows users **PuTTY** should be installed and for Linux users **minicom** should be installed.

1.2 Booting Linux on the ZedBoard

Now let's refer to Figure 1.1 and take a look at how Linux is booted on the ZedBoard. When the ZedBoard is powered on, the processor on the Zynq device loads into RAM a small, fixed number of bytes from the start of a FAT formatted SD-Card. This small amount of data contains the First Stage Boot Loader (FSBL) for the Zynq device. The purpose of the FSBL is to load the Primary Boot Loader, which in our case is Das U-Boot, which is capable of booting the Linux kernel. Once the FSBL is loaded, the processor firmware sets the program counter to the entry point of the FSBL. The FSBL then loads and executes Das U-Boot. In order to successfully boot Linux, we need the Linux kernel itself, the Root File System (containing all our programs and configurations) and the Device Tree (containing information about all the peripherals we have on the ZedBoard). U-Boot will find these files on the SD-Card and load them into memory. U-Boot then sets

the program counter to the entry point of the Linux kernel. Now the Linux kernel will configure and load the correct device drivers according to the Device Tree and then mount the root file system. Then the init process (found in the Root File System) is executed and an SSH server is started.

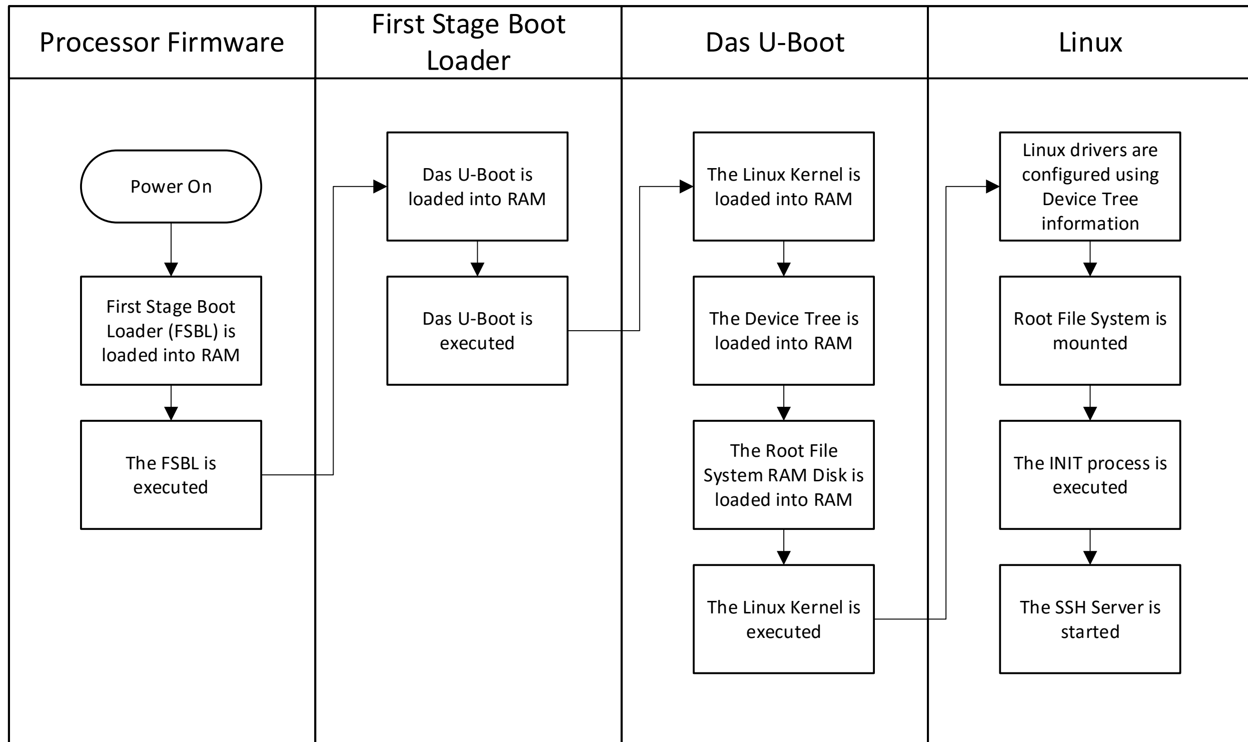

Figure 1.1: Boot Sequence for Linux on the Zynq Device

1.2.1 Preparing the ZedBoard and the SD-Card

First we can download the pre-built images provided by Xilinx. We will be using the 2015.4 release for the ZedBoard; `http://www.wiki.xilinx.com/file/detail/2015.4-zed-release.tar.xz`

Once downloaded and extracted, you will see the following files;

- **fsbl.elf**: FSBL stands for "First Stage Boot Loader" and is the first program that will be executed by the processor. In our case, the main purpose of the FSBL is to load Das U-Boot.
- **u-boot.elf**: U-Boot is a boot loader capable of booting Linux. U-Boot itself is loaded by the FSBL.
- **devicetree.dtb**: The device tree is a file that is read by the Linux Kernel as it is starting up. It describes which peripheral devices are available to the system and how to access them (ie. where in the address space does its control registers lie).
- **boot.bin**: The boot.bin is the file which contains both the *fsbl.elf* and *u-boot.elf*. This is the actual file that is read from the SD-Card, whereas the separate fsbl.elf and u-boot.elf files are supplied for completeness.
- **uImage**: The uImage is the compiled Linux kernel wrapped with u-boot headers and is loaded by U-Boot.
- **uramdisk.image.gz**: This file is a filesystem image which will be loaded by the Linux kernel as a ramdisk and will act as the root file system.

Prepare your SD-Card by formatting it to the FAT-32 format. Then copy **boot.bin**, **uImage**, **uramdisk.image.gz** and **devicetree.dtb** to the SD-Card. Now the SD-Card is ready to be inserted into the ZedBoard.

Now let's make sure the ZedBoard is configured for booting with the SD-Card. Make sure the jumpers are set to the correct positions;

- JP6 - Shorted
- JP7 - GND
- JP8 - GND
- JP9 - 3V3
- JP10 - 3V3
- JP11 - GND

Connect the ZedBoard to your network (ie. to your router/switch or directly to your computer) using an Ethernet cable and connect the UART port to your computer using USB. Now we are ready to boot Linux on the ZedBoard.

1.2.2 Booting the Zynq Device

Power on the ZedBoard and open a serial terminal in order to monitor the bootup process. On Windows we can use PuTTY, select **serial** under Connection Type, set the Speed (baud rate) to **115200**. On Linux we can use minicom by entering the following command in the terminal window;

```
sudo minicom -D /dev/ttyACM0/ -b 115200
```

If you see an error saying that the device was not found, check that you have entered the correct COM port on Windows or the correct tty device on Linux. Otherwise, once we're in our serial terminal, power off and power on the ZedBoard, so we can see the U-Boot debug messages from the beginning. If all goes well and the bootup has completed, we will see the prompt;

```
zedboard-zynq7 login:
```

We can type **root** to log in and then we are in the Linux root user shell, we can issue any standard Linux shell commands here. We can explore the filesystem using *ls* and *cd* for example.

1.2.3 Connecting to the ZedBoard over SSH

Interacting with Linux over a UART serial connection is not stable, we should instead log in using SSH over the network. If you have a DHCP server running on your network, that is, you have your ZedBoard plugged into your router, an IP address should already be assigned to your board. You can find your IP address by typing this command in your serial terminal;

```
ifconfig
```

If the Zynq device can not find a DHCP server on your network (ie. you don't see an IP address under ifconfig), then we must set up a static address. If the ZedBoard is plugged into your computer directly, then we must setup a static address for both the Ethernet adapter on your computer and on the ZedBoard. For Linux users, we can simply issue the following command on the host computer;

```
ifconfig eth0 192.168.33.1
```

Windows users can use the Network Connections utility under the control panel. Now in our serial terminal, we set a static address for the Ethernet adapter on the ZedBoard;

```
ifconfig eth0 192.168.33.2
```

Once we have obtained the IP address, we can try connecting to it from our host computer. On Windows, we can use PuTTY again in SSH mode and on Linux we can issue the following command;

```
ssh root@{ZedBoard_IP_Address}
```

We should be able to log in without a password and be greeted with a standard Linux shell. ***NOTE: To permanently assign a static IP address to the Zynq device, you must mount and edit the Root File System RAM disk on a Linux host. Please check section 2.6 for details on how to modify the root file system***

1.3 Configuring the FPGA at Runtime under Linux

1.3.1 Creating a Simple GPIO Project Using Vivado

Let's open Vivado and create a Project called "PS_LED", targeting the ZedBoard. Create a new Block Design called "system" and add the "ZYNQ7 Processing System" IP block. Now, run block automation leaving all the settings at their defaults. Double click the IP block to access the customization menu, then click on "MIO Configuration", then "I/O Peripherals", "GPIO", check "EMIO GPIO", set the width to 8 (Note that the maximum width is 64) and click "OK". Now a new port will show up on the block diagram called "GPIO_0" and if we expand this bundled port we will see it is made up of an 8-bit wide input port, an 8-bit wide output port and an 8-bit wide tri-state port. In fact this is very similar to the GPIO interface found on the AXI GPIO IP block, except it is implemented on the hard processor side of the Zynq device and thus does not take up any reconfigurable resources on the FPGA. Let's connect it up to some LEDs and switches. Right click on both `GPIO_I` and `GPIO_O` and select *Make External*. This will create two external ports, `GPIO_I` and `GPIO_O`. Now let's configure these ports so that Vivado knows to connect them to the LEDs and switches. Select the `GPIO_I` port and under External Port Properties, change the NAME field to **sws_8bits**. Likewise, for the `GPIO_O` port change the NAME field to **leds_8bits**. We must now assign physical I/O pins on the FPGA to our led and switch signals, by creating a constraints file. Click on *Add Sources*, *Add or create constraints* and create a new constraints file called **ZedBoard_Master.xdc**. In this file add the following lines;

```
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds_8bits[0]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[7]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[6]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[5]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[4]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[3]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {sws_8bits[0]}]
set_property PACKAGE_PIN U14 [get_ports {leds_8bits[7]}]
```

```
set_property PACKAGE_PIN U19 [get_ports {leds_8bits[6]}]
set_property PACKAGE_PIN W22 [get_ports {leds_8bits[5]}]
set_property PACKAGE_PIN V22 [get_ports {leds_8bits[4]}]
set_property PACKAGE_PIN U21 [get_ports {leds_8bits[3]}]
set_property PACKAGE_PIN U22 [get_ports {leds_8bits[2]}]
set_property PACKAGE_PIN T21 [get_ports {leds_8bits[1]}]
set_property PACKAGE_PIN T22 [get_ports {leds_8bits[0]}]
set_property PACKAGE_PIN M15 [get_ports {sws_8bits[7]}]
set_property PACKAGE_PIN H17 [get_ports {sws_8bits[6]}]
set_property PACKAGE_PIN H18 [get_ports {sws_8bits[5]}]
set_property PACKAGE_PIN H19 [get_ports {sws_8bits[4]}]
set_property PACKAGE_PIN F21 [get_ports {sws_8bits[3]}]
set_property PACKAGE_PIN H22 [get_ports {sws_8bits[2]}]
set_property PACKAGE_PIN G22 [get_ports {sws_8bits[1]}]
set_property PACKAGE_PIN F22 [get_ports {sws_8bits[0]}]
```

NOTE: Whitespace matters! Make sure there is no whitespace between the port name and the indices/brackets. If you get critical warnings during synthesis, please double check the whitespace!

Finally, connect `M_AXI_GP0_ACLK` to `FCLK_CLK0`, or else the project will fail to synthesize. Your project should now match Figure 1.2.

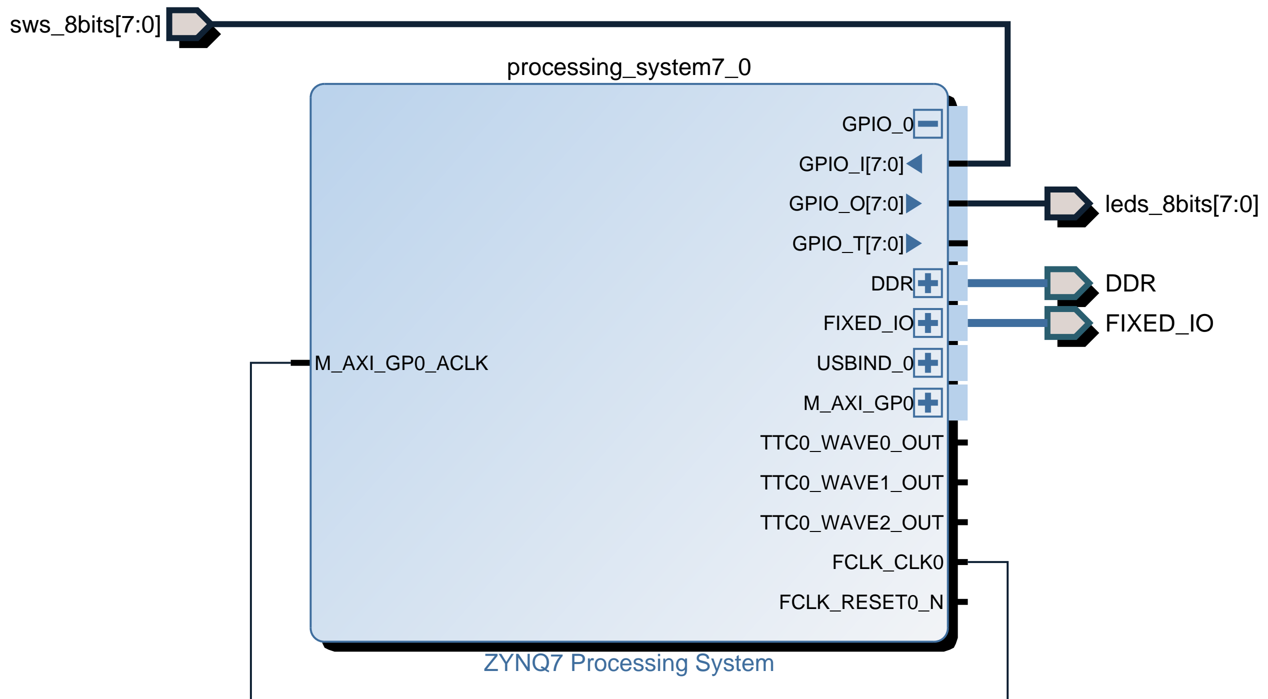

Figure 1.2: Final Block Design

Now, we can create an HDL wrapper for the block design by right clicking on **system.bd** in *Design Sources* and selecting *Create HDL Wrapper*. Choose *Let Vivado manage wrapper and auto-update* and click *OK*. Now we can implement the design and generate the bitstream. Once that is done, we can confirm that we have used no reconfigurable logic resources (ie. flip-flops, LUTs) other than routing resources for this project, by

checking the *Utilization - Post Implementation* pane in the *Project Summary* tab. We can find the generated bitstream here;

```
{Project Parent}/PS_LED/PS_LED.runs/impl_1/system_wrapper.bit
```

1.3.2 Configuring the FPGA

Now let's upload the bitstream file to the Linux file-system on the ZedBoard. Windows users can use a program like WinSCP and Linux users can simply enter the following command;

```
scp {Project Parent}/PS_LED/PS_LED.runs/impl_1/system_wrapper.bit
   root@{ZedBoard_IP_Address}:.
```

The transfer should be very quick and once it's done, you will be able to see the bitstream file in your `/home/root` directory. To configure the FPGA, simply run the following command in our SSH terminal;

```
cat system_wrapper.bit > /dev/xdevcfg
```

The **xdevcfg** device file is the Device Configuration Interface, which allows the Zynq Processing System to configure the FPGA (ie. Programmable Logic). Like other Linux device files, we can access **xdevcfg** using standard Input/Output system calls, such as *Open*, *Read*, *Write* and *Close*. The command above dumps the contents of our bitstream onto the xdevcfg device which configures the FPGA. You should now see the **DONE LED** light up, signalling that the FPGA has been configured.

1.4 Interfacing with the FPGA using GPIO under Linux

1.4.1 Creating a C/C++ Project in Xilinx SDK

Xilinx has provided an easy way for compiling programs for the Zynq Device, through the Xilinx SDK. Let's open the XSDK and create a new application project;

```
File -> New -> Application Project
```

Set the *Project name* to **GPIO_SW_LED**, set *OS Platform* to **linux**, make sure the *Processor Type* is **ps7_cortexa9**, *Language* as **C** and click next. Select **Linux Empty Application** from *Available Templates* and click finish. Now in the Project Explorer, an empty project has been created, add a new file called **main.c** in the **src** directory and add the following code to it;

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

// The first EMIO GPIO pin is number 960 under Linux on the ZedBoard
// GPIO base address is 906 and EMIO starts 54 after that.
// To check the GPIO base address run;
// "cat /sys/class/gpio/gpiochip906/base"
int main() {
    int fd;
    // export EMIO GPIO 0
    fd = open("/sys/class/gpio/export", O_WRONLY);
```

```
        write(fd, "960", 3);
        close(fd);
        // Set EMIO GPIO 0 to output mode
        fd = open("/sys/class/gpio/gpio960/direction", O_WRONLY);
        write(fd, "out", 3);
        close(fd);
        // Set EMIO GPIO 0 to high, turn on LED
        fd = open("/sys/class/gpio/gpio960/value", O_WRONLY);
        write(fd, "1", 1);
        close(fd);
        // unexport EMIO GPIO 0
        fd = open("/sys/class/gpio/unexport", O_WRONLY);
        write(fd, "960", 3);
        close(fd);
        // export EMIO GPIO 1
        fd = open("/sys/class/gpio/export", O_WRONLY);
        write(fd, "961", 3);
        close(fd);
        // Set EMIO GPIO 1 to input mode
        fd = open("/sys/class/gpio/gpio961/direction", O_WRONLY);
        write(fd, "in", 2);
        close(fd);
        // Read EMIO GPIO 1, read switch
        fd = open("/sys/class/gpio/gpio961/value", O_RDONLY);
        char switch_value;
        read(fd, &switch_value, 1);
        printf("Switch[1]:␣%c\n", switch_value);
        close(fd);
        // unexport EMIO GPIO 1
        fd = open("/sys/class/gpio/unexport", O_WRONLY);
        write(fd, "961", 3);
        close(fd);
        return 0;
    }
```

This program exports (reserves) GPIO 960, which is the first EMIO GPIO signal, then the program sets the direction of GPIO 960 to output and writes a one to turn LED0 on, then GPIO 960 is unexported (freed). The program then exports GPIO 961, sets the direction to input and reads the input value of EMIO GPIO 1 (which we connected to SW1). Upon saving the file, the SDK will attempt to compile the project. Under the **Binaries** directory within the project, we can see a GPIO_SW_LED.elf file. This is the compiled program which we want to execute on the ZedBoard. Navigate to the directory containing GPIO_SW_LED.elf and execute the following command in order to upload the file to the ZedBoard;

```
    scp ./GPIO_SW_LED.elf root@{ZedBoard_IP_Address}:.
```

Windows users may use WinSCP. Now in our Zynq shell we should be able to see **GPIO_SW_LED.elf** by running **ls** and execute it by running **./GPIO_SW_LED.elf**. We should now see LED0 light up and the position of SW1 printed in our terminal. Note that as soon ass we export any GPIO pins, the output of a particular pin is latched to the input of that pin. That is why LED1 lights up when we export the GPIO while SW1 is in the on position.

Chapter 2

Building the Boot System from Scratch (Advanced)

2.1 Introduction

In this section we will manually build the boot system for the Zynq device. The motivation for doing this may be that you want to boot the Linux kernel over the network, compile the Linux kernel with different parameters, compile your own custom driver along with the kernel or just for fun. We will;

- Build the Device Tree Compiler
- Build Das U-Boot
- Build the Linux kernel
- Generate the First Stage Boot Loader using the Xilinx SDK
- Create the boot.bin by combining the FSBL with U-Boot

Lets first set up a directory for our project;

```
mkdir -p ~/ZedBoard_Linux/Boot_System
cd ~/ZedBoard_Linux/Boot_System
```

2.2 Building the Device Tree Compiler

Now we need the Device Tree Compiler required to build Das U-Boot. We can obtain the DTC from the Xilinx repositories;

```
cd ~/ZedBoard_Linux/Boot_System
git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
```

After it has been downloaded, we can build the DTC;

```
cd dtc
make
```

Let's add this to our PATH variable so that the U-Boot build system can access it;

```
export PATH=$PATH:$PWD
```

2.3 Building Das U-Boot

Let's start by downloading the U-Boot source code from the Xilinx Repositories;

```
cd ~/ZedBoard_Linux/Boot_System
git clone https://github.com/Xilinx/u-boot-xlnx.git
```

Navigate to the U-Boot directory;

```
cd u-boot-xlnx
```

When U-Boot starts up, it will copy the Linux Kernel from external memory to RAM, For the Zynq Device, it is possible to load the Linux kernel from these locations; MMC (MultiMediaCard), eMMC (Embedded MultiMediaCard), SD Card, NOR Flash, QSPI Flash, NAND Flash, JTAG or through TFTP over the network. The build method for booting Linux from SD-Card and over the network is described below. If you wish to boot solely from the SD-Card (default behaviour) then skip the following subsection.

2.3.1 Configure for Booting Over the Network (Optional)

If you wish to configure U-Boot to fetch Linux from a TFTP server, first we must modify the default U-Boot configuration for the Zynq device. Open the following file with your favourite text editor;

```
vim ~/ZedBoard_Linux/Boot_System/u-boot-xlnx/include/configs/zynq-
   common.h
```

Search for the following line of text;

```
"sdboot=if mmcinfo; then " \
```

The indented section below it specifies which commands U-Boot will execute once it has loaded from a MultiMediaCard, in our case, the SD-Card. In fact, the First-Stage-Boot-Loader will inform U-Boot exactly where it booted from. Currently, it will look for the operating system kernel, device tree and the root file system ramdisk on the SD-Card using the **load mmc 0** command. They will be loaded into specific addresses in RAM designated by the load address variables and booted using the **bootm** command;

```
"sdboot=if mmcinfo; then " \
    "run uenvboot; " \
    "echo Copying Linux from SD to RAM... && " \
    "load mmc 0 ${kernel_load_address} ${kernel_image} && " \
    "load mmc 0 ${devicetree_load_address} ${devicetree_image} && " \
    "load mmc 0 ${ramdisk_load_address} ${ramdisk_image} && " \
    "bootm ${kernel_load_address} ${ramdisk_load_address} ${
       deivcetree_load_address}; " \
"fi\0" \
```

To boot over the network, we must change it to the following;

```
    "sdboot=if mmcinfo; then " \
        "run uenvboot; " \
        "echo Copying Linux from SD to RAM... && " \
        "tftpboot ${kernel_load_address} 10.10.70.101:${kernel_image} && "
            \
        "tftpboot ${devicetree_load_address} 10.10.70.101:${
           devicetree_image} && " \
        "tftpboot ${ramdisk_load_address} 10.10.70.101:${ramdisk_image} &&
            " \
        "bootm ${kernel_load_address} ${ramdisk_load_address} ${
           deivcetree_load_address}; " \
    "fi\0" \
```

As you can see, we have replaced the **load mmc 0** command with **tftpboot** and prefixed our kernel, devicetree and ramdisk image variables with our TFTP server address. This commands U-Boot to fetch the images from the root directory on our TFTP server.

2.3.2 Building U-Boot

Now we can build U-Boot, navigate to the U-Boot directory;

```
    cd ~/ZedBoard_Linux/Boot_System/u-boot-xlnx
```

We configure the build system to build for the Zedboard and build;

```
    make zynq_zed_config
    make
```

If you have another board (ie. not the ZedBoard), check the following directory to see if there is a configuration file for your board;

```
    ls ~/ZedBoard_Linux/Boot_System/u-boot-xlnx/include/configs
```

Once the make completes, an ELF file **u-boot** will be generated. Copy this to the folder we made to contain our completed files and append the .elf extension;

```
    cp ./u-boot ~/ZedBoard_Linux/Boot_System/Complete/u-boot.elf
```

The **mkimage** tool needed for wrapping the Linux kernel with u-boot headers will be generated in the tools directory, so we need to add this to our path;

```
    export PATH=$PATH:$PWD/tools
```

2.4 Building the Linux Kernel

Let's download the Linux kernel source code;

```
    cd ~/ZedBoard_Linux/Boot_System
    git clone https://github.com/Xilinx/linux-xlnx.git
```

Now we can finally build the Linux kernel.

```
cd  linux-xlnx
make ARCH=arm xilinx_zynq_defconfig
```

This configures the build config to the default settings that Xilinx has set for the Zynq device. If you wish to configure this manually, you may run;

```
make ARCH=arm menuconfig
```

We can now compile the kernel;

```
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

This will generate an uImage which is the compiled linux kernel wrapped with an U-Boot header. You can find and copy it from the following directory;

```
cp ./arch/arm/boot/uImage ~/ZedBoard_Linux/Boot_System/Complete
```

Generated along with the kernel image is the device tree. We need to compile it into binary format;

```
dtc -o ~/ZedBoard_Linux/Boot_System/Complete/devicetree.dtb -O dtb ./
    arch/arm/boot/dts/zynq-zed.dts
```

2.5 Building the First Stage Boot Loader and Creating BOOT.bin

2.5.1 Building the FSBL

Let's open the Xilinx SDK by issuing the following command in the terminal;

```
xsdk
```

When the XSDK GUI opens, we will create a new FSBL application project;

```
File -> New -> Application Project
```

Set the project name to be **FSBL**. Then, under the Hardware Platform drop down menu, select **zed_hw_platform**. Make sure the OS Platform is set to **Standalone**. Now click next, select **Zynq FSBL** under Available Templates and click Finish.

Now a new FSBL project will be created and in the **Binaries** directory, we can find the FSBL.elf and move it our **Complete** directory.

2.5.2 Creating BOOT.bin

In Xilinx SDK, click on *Xilinx Tools*, then *Create Boot Image*. Under *Boot Image Partitions* click *Add* and add the **FSBL.elf** file. Then click *Add* again, add the u-boot.elf file and then press create the BOOT.bin image. Copy this to our **Complete** directory.

2.6 Modifying the Root File System

In this section, we will use the default Root File System RAM disk which can be downloaded as a part of the Xilinx official release here; `http://www.wiki.xilinx.com/file/detail/2015.4-zed-release.tar.xz`

We must unwrap the U-Boot headers from the image;

```
dd if=uramdisk.image.gz bs=64 skip=1 of=ramdisk.image.gz
```

We can extract the Root File System image using gzip so we can modify it;

```
mkdir tmp_mnt
gunzip ramdisk.image.gz | sudo sh -c 'cd tmp_mnt/ && cpio -i'
cd tmp_mnt
```

After we have finished modifying, we can unmount and compress the image;

```
sh -c 'cd tmp_mnt/ && sudo find . | sudo cpio -H newc -o' | gzip -9 >
   ramdisk.image.gz
```

Now we have to wrap the compressed image with the U-Boot headers again;

```
mkimage -A arm -T ramdisk -C gzip -d ramdisk.image.gz uramdisk.image.
   gz
```

2.7 Setting Up the TFTP Server (Optional)

This step only applies if you want to boot Linux over the network and you have compiled U-Boot for booting over the network. If you have not installed a TFTP server then you must do so now, we will be using the tftpd-hpa package;

```
sudo apt-get install tftpd-hpa
```

The TFTP server will serve any files placed in the following directory;

```
ls -l /var/lib/tftpboot
```

We need to copy our compiled Linux kernel, the device tree and the root file system ramdisk to this location;

```
cp ~/ZedBoard_Linux/Boot_System/Complete/uImage /var/lib/tftpboot
cp ~/ZedBoard_Linux/Boot_System/Complete/devicetree.dtb /var/lib/
   tftpboot
cp ~/ZedBoard_Linux/Boot_System/Complete/uramdisk.image.gz /var/lib/
   tftpboot
```

To start the TFTP server, simply issue;

```
sudo service tftpd-hpa restart
```

Now when U-Boot is loaded on the ZedBoard, U-Boot will fetch files needed to boot Linux from the TFTP server.