



Embedded Linux® Hands-on Tutorial for the ZYBO™

Revised July 17, 2014

This tutorial uses Vivado under Linux. If stand-alone Vivado is used and Vivado is not installed under Linux, Vivado cross-compile tools should be installed under Linux to compile source code for Xilinx. Those tools have a prefix arm-xilinx-linux-gneabi- to the standard names for the GCC tool chain. If SDK 2018.3 is used, source its settings by `$source /opt/Xilinx/SDK/2018.3/settings64.sh`.

Overview

The purpose of this document is to provide step-by-step instructions for customizing your hardware, compiling the Linux Kernel, and writing driver and user applications. This documentation intends to integrate knowledge and skills in FPGA logic circuit design, standalone software programming, Linux operating system and user application development, and apply them to the ZYBO. We will start from the *ZYBO Base System Design* (available on the ZYBO product page of the Digilent website). The system architecture for the *ZYBO Base System Design* is shown in Fig. 1.

In the *ZYBO Base System Design*, we connect UART1 to USB-UART, SD0 to the SD Card Slot, USB0 to the USB-OTG port, Enet0 to the Giga-bit Ethernet Port, and Quad SPI to the on-board QSPI Flash. These cores are hard IPs inside the Processing System (PS) and connect to on-board peripherals via Multiplexed I/O (MIO) pins. The use of PS GPIO is connected to Btn 4 and 5. In the Programmable Logic (PL), we have an HDMI TX Controller, VDMA, and GPIO IP cores to talk to the ADV7511 HDMI transmitter chip and I2S and GPIO IP cores for ADAU1761 audio codec. More details of the hardware design can be found in the documentation inside the *ZYBO Base System Design* package.

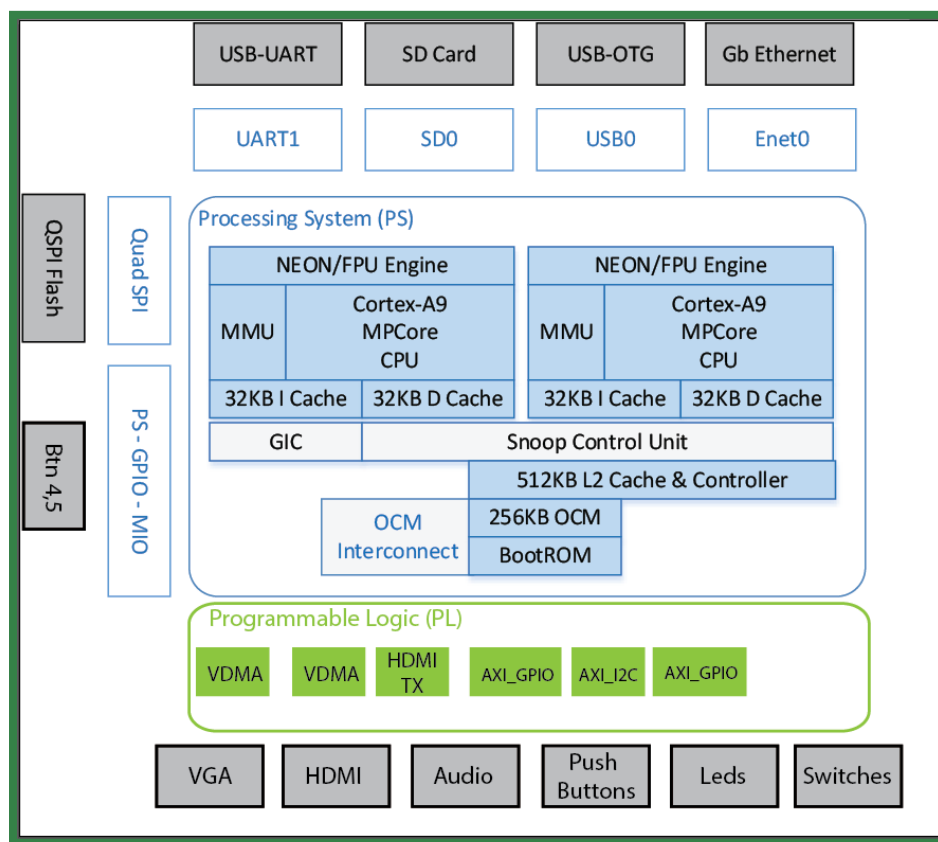


Figure 1. Reference Basic Hardware System Architecture for ZYBO.

In this tutorial, we are going to detach the LEDs from the AXI GPIO core and implement our own myLed core for it in PL, as shown in Fig. 2. We will then add our own LED controller into the device tree, write a driver for it, and develop user applications to control the status of the LEDs.

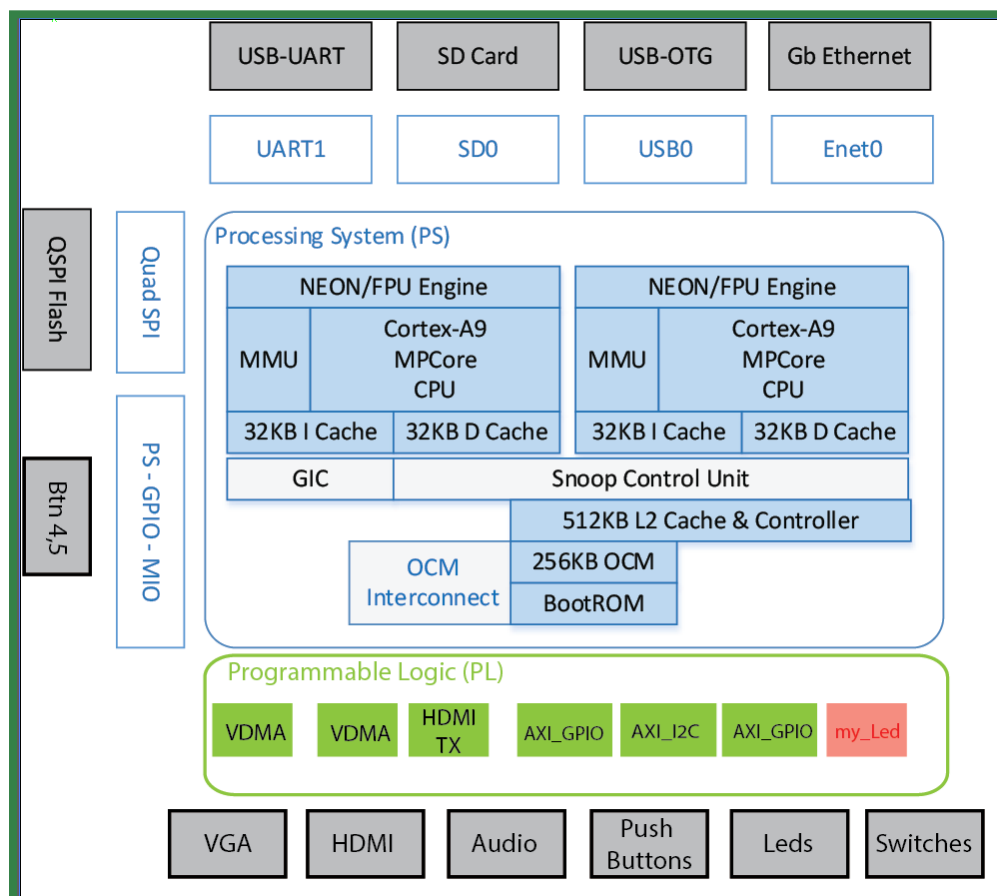


Figure 2. Hardware System Architecture of the system we are going to implement in this Tutorial.

Before going through this tutorial, we recommend that you read *Getting Started with Embedded Linux - ZYBO*. You can follow this tutorial with the *Embedded Linux Development Guide* (available on the Diligent website Embedded Linux Page). The guide will provide you with the knowledge you may need in each step of the development.

In this tutorial, we are going to use Xilinx® Vivado™ 2014.1 WebPACK™ in a Linux environment. All of the screenshots and codes are done using Vivado Design Suite 2014.1 in CentOS 6 x86_64.

That's it for the background information on this tutorial, now it's time to get our hands dirty with some real design!

1 Hardware Customization

1.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx website [Download Page](#).
- *ZYBO Base System Design*: available at the Digilent website on the [ZYBO Page](#).

1.2 Instructions

1. Download the *ZYBO Base System Design* from the Digilent website and unzip it into our working directory, as in Fig. 3 (our working directory is named **tutorial** throughout this document). For more information on the hardware design, please refer to *Project Guide* under doc folder.

```
[kfranz@localhost Tutorial]$ unzip /home/kfranz/Downloads/zybo_base_system.zip
Archive:  /home/kfranz/Downloads/zybo_base_system.zip
  inflating: zybo_base_system/ProjectGuide.txt
   creating: zybo_base_system/sd_image/
  inflating: zybo_base_system/sd_image/B00T.bin
   creating: zybo_base_system/source/
   creating: zybo_base_system/source/ise/
   creating: zybo_base_system/source/ise/hw/
   creating: zybo_base_system/source/ise/hw/data/
  inflating: zybo_base_system/source/ise/hw/data/ps7_constraints.ucf
```

Figure 3. Unzip the ZYBO_Base_System.

2. Source Vivado 2014.1 settings and open the design with Vivado Design Suite. You will see the Vivado window pop up as shown in Fig. 4.

Note: There are four settings files available in the Vivado toolset: settings64.sh for use on 64-bit machines with bash; settings32.sh for use on 32-bit machines with bash; settings32.csh for use on 32-bit machines with C Shell; and settings64.csh for use on 64-bit machines with C Shell. **Change all the version numbers to 2014.4**

```
[kfranz@localhost Tutorial]$ source /opt/Xilinx/Vivado/2014.1/settings64.sh
[kfranz@localhost Tutorial]$ vivado zybo_base_system/source/vivado/hw/zybo_bsd/z
ybo_bsd.xpr

***** Vivado v2014.1 (64-bit)
***** SW Build 881834 on Fri Apr  4 14:00:25 MDT 2014
***** IP Build 877625 on Fri Mar 28 16:29:15 MDT 2014
** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

start_gui
```

Open the project with Vivado's GUI

Figure 4. Open the Project.

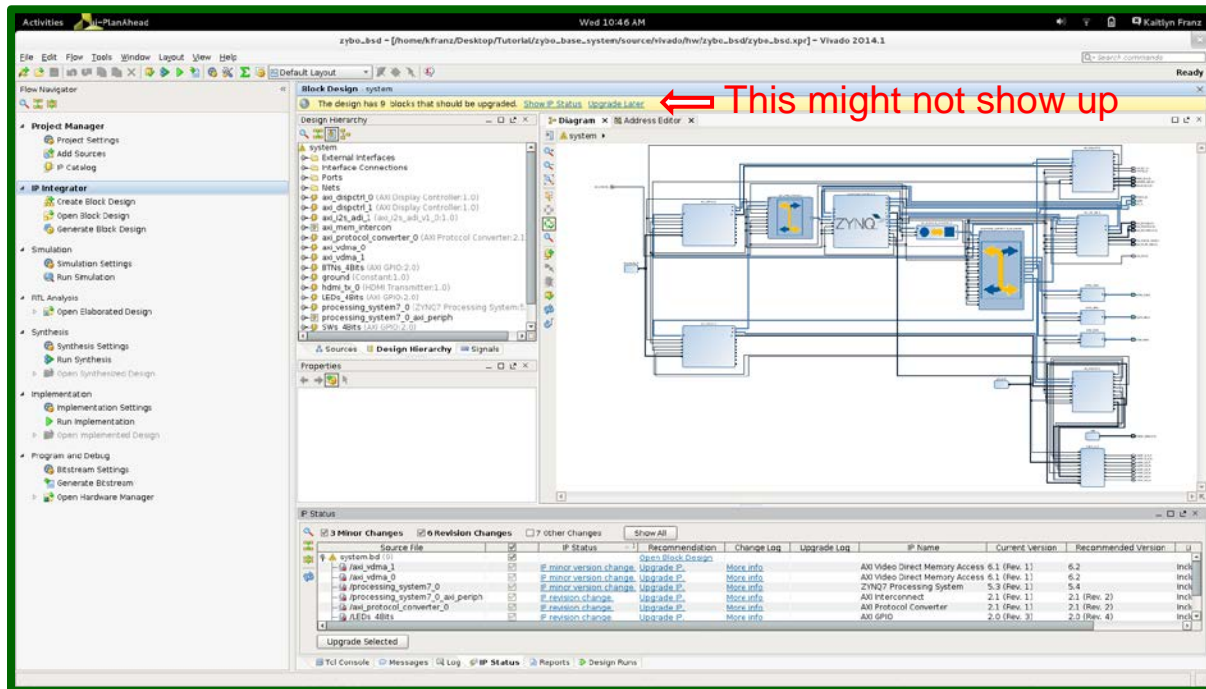


Figure 5. Vivado platform studio GUI.

- We are going to detach LEDs from the GPIO core in the PS first. So we need to click on the IP integrator and open the **Block Diagram** as shown in Fig. 5. Then we need to delete the current LED IP as shown in Fig. 6. We will handle the modification of external pin location configuration (**xdc** file) in later steps.

Note: In Fig. 6 there is a yellow bar indicating the need for an upgrade. To upgrade, hit **show IP status**, make sure all are selected and hit **Upgrade Selected**.

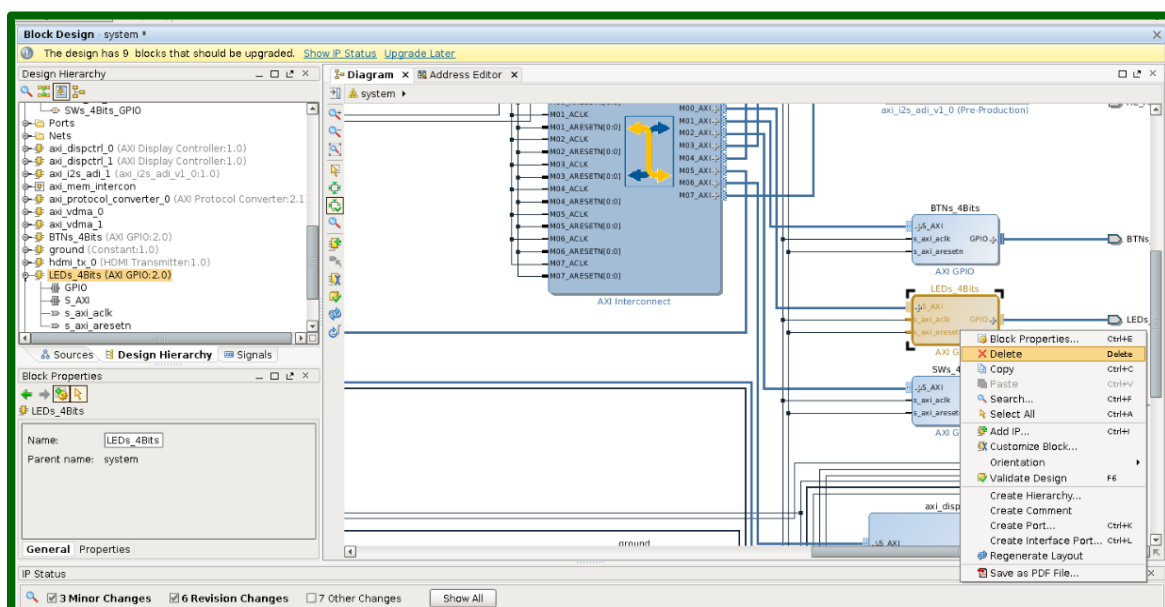


Figure 6. Delete existing LED IP.

- This step is only required in Linux**
- (Vivado 2014.1 only) Before we can start implementing our myLed IP Core, we need to name the vendor that will automatically be applied in the IP packager. In Vivado 2014.1, this is not automatically done for the user. To do this, first go to the **Project Settings** under **Project Manager** on the left side of the window (Fig. 7) and the project settings window will pop up. In the Project Settings window, select **IP** (Fig. 8). Notice that the vendor is chosen as “(none)”, this will cause a Vivado internal exception. You can name the Vendor whatever you like (Fig. 9).

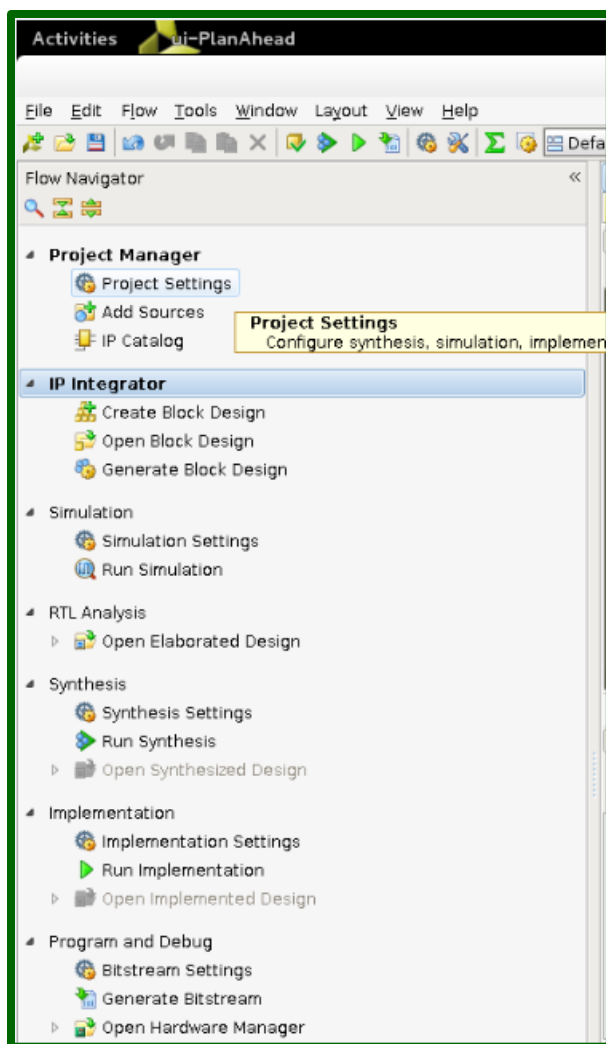


Figure 7. Project settings.

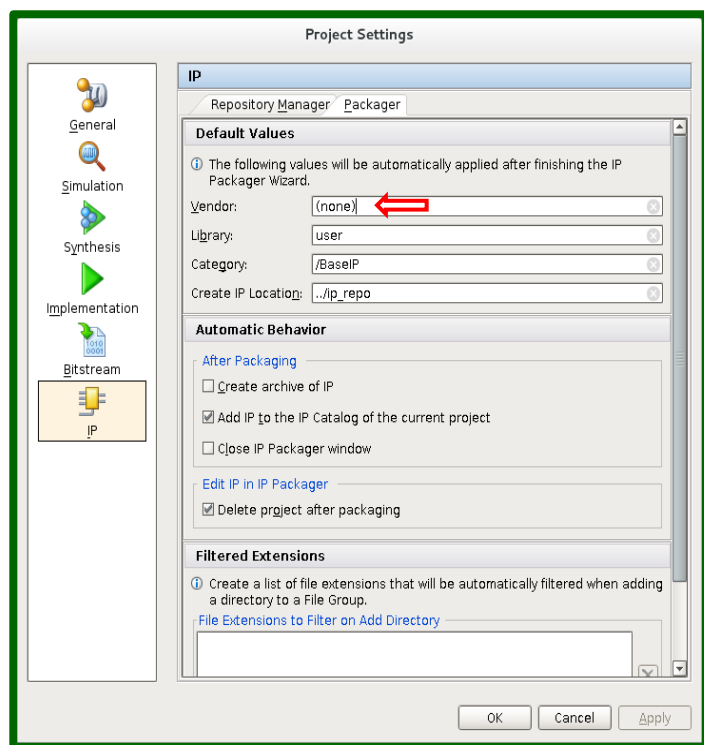


Figure 8. Unnamed vendor.

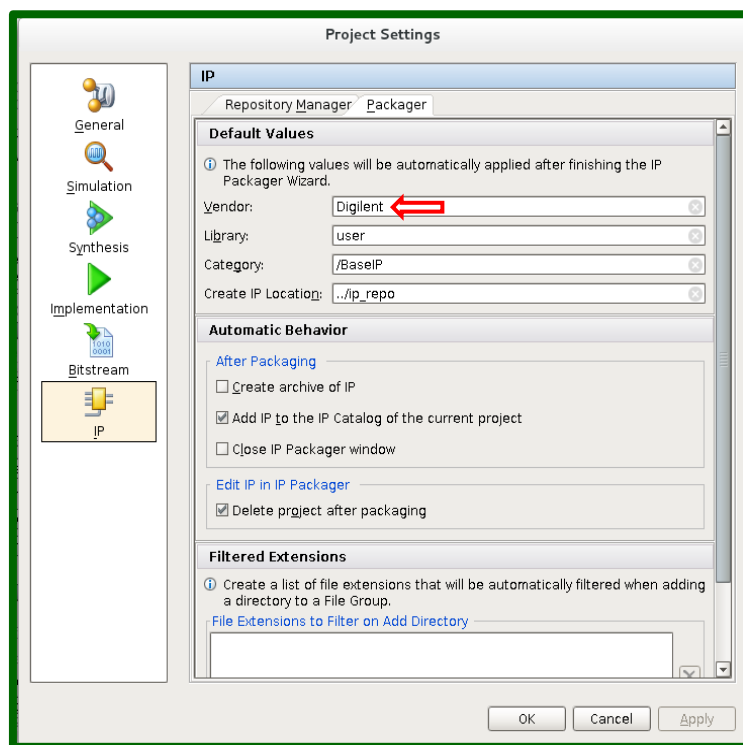


Figure 9. Named vendor.

- Now we can start implementing our myLed IP Core. Click **Tools -> Create and Package IP...** from the menu (as shown in Fig. 10). The Create and Package New IP window will pop up (as shown in Fig. 11), Click **Next**. In the next window, name the new IP and click next again (Fig. 12).

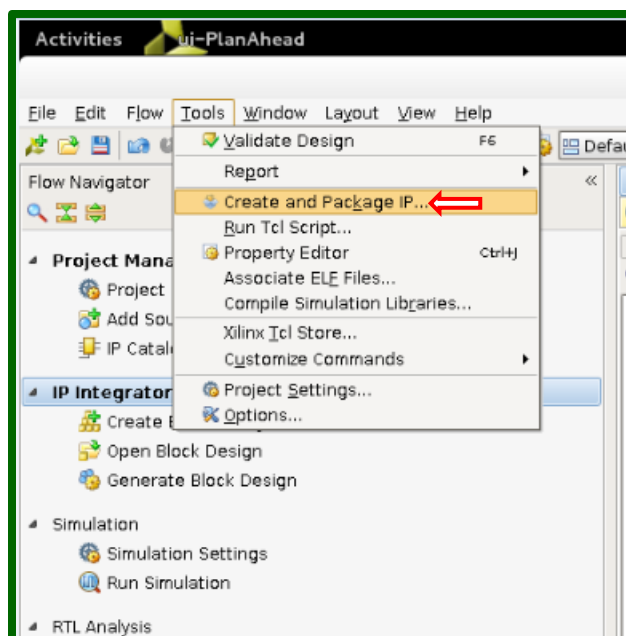


Figure 10. Create and Package IP.

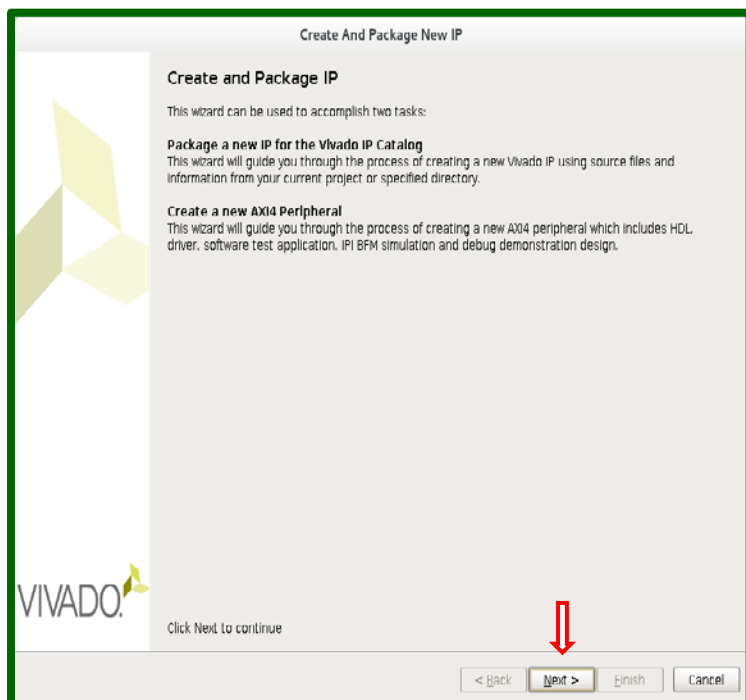


Figure 11. IP Options.

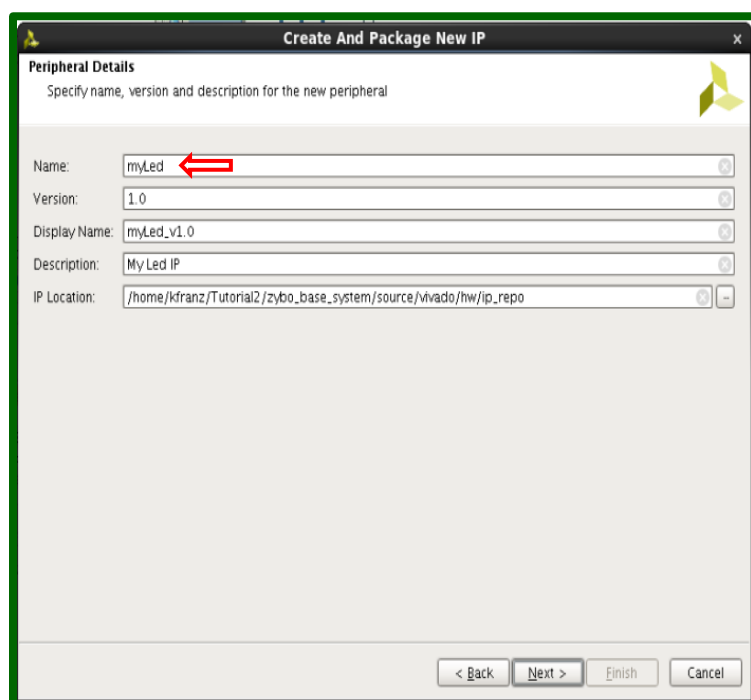


Figure 12. Peripheral Details.

- The next window will be the Add Interfaces Window. This will create the AXI4 Interface for the myLed peripheral (Fig. 13). Make sure the interface type is **Lite**, the mode is **Slave**, the data width is **32 bits** and the number of registers is **4**. Change the Name to **S_AXI** rather than S00_AXI. We only need 1 register but the minimum we can select is 4. Click next to proceed.

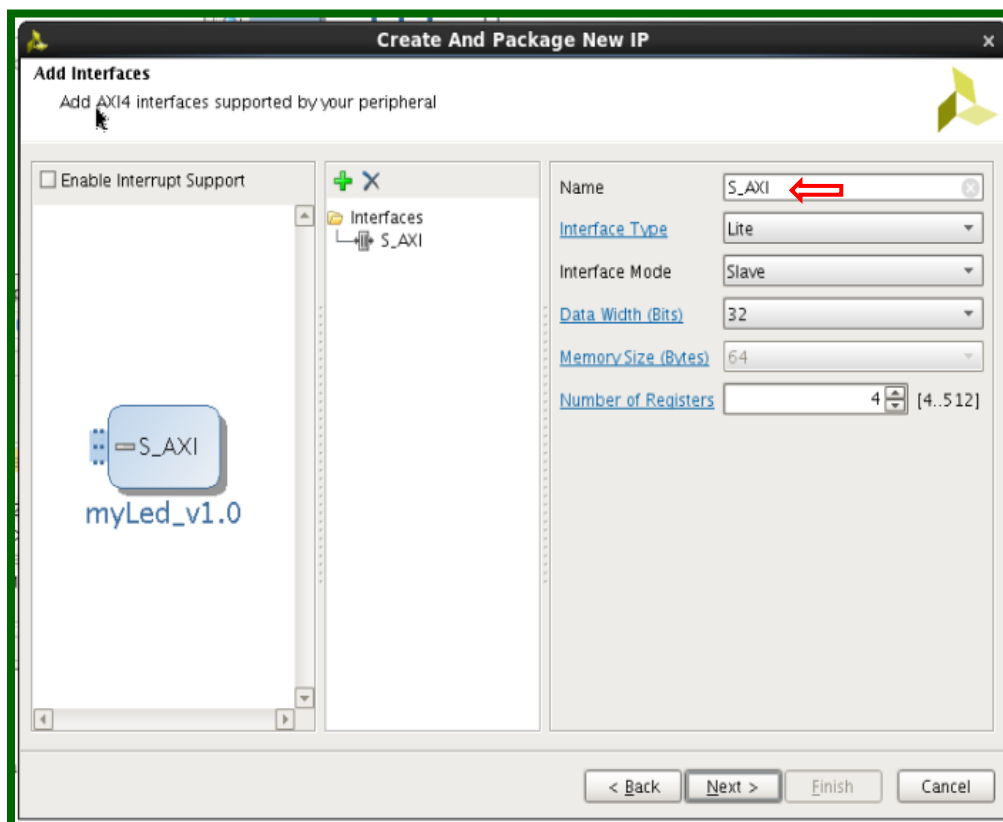


Figure 13. Add Interface

7. The next window will prompt the finishing steps to create the IP (Fig. 14). Change the Radio button to select **Edit IP** and hit finish. We need to add user logic to the IP so that our slave is connected to the LED output.

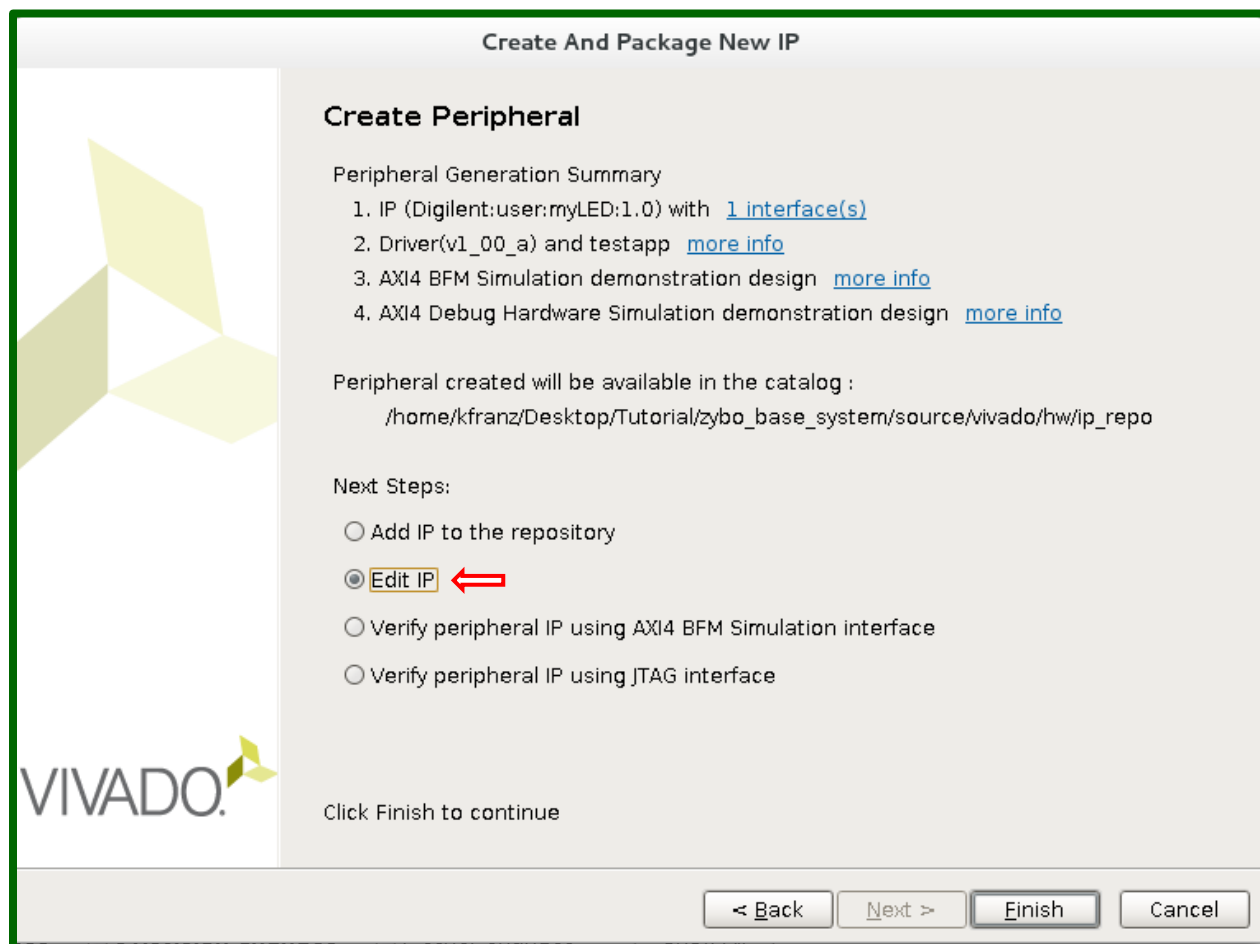


Figure 14. Edit IP.

8. After selecting finish, the Create and Package IP window will disappear and the next window you will see is the edit_myLed window (Fig. 15). This is where we will add our user logic.

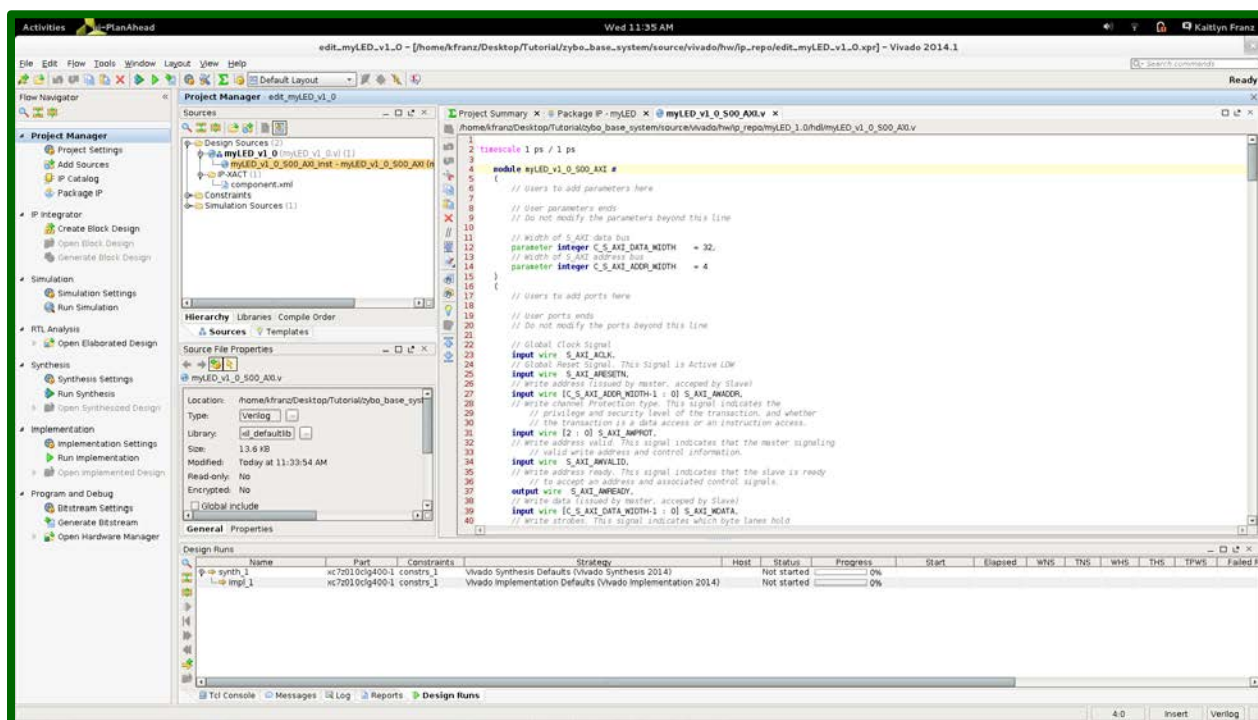


Figure 15. Edit_myLed.

9. In the Project Manager, click the circle next to myLed_v1_0 and highlight **myLed_v1_0_S_AXI** (Fig. 16). This contains the user logic inside of the myLed IP. We need to add two lines of code to complete the user logic for this module. First, we need to create a user port called led (Fig. 17). Next, we need to connect the internal slave to this user port. We will connect slv_reg0[3:0] as we have four LEDs (Fig. 18).

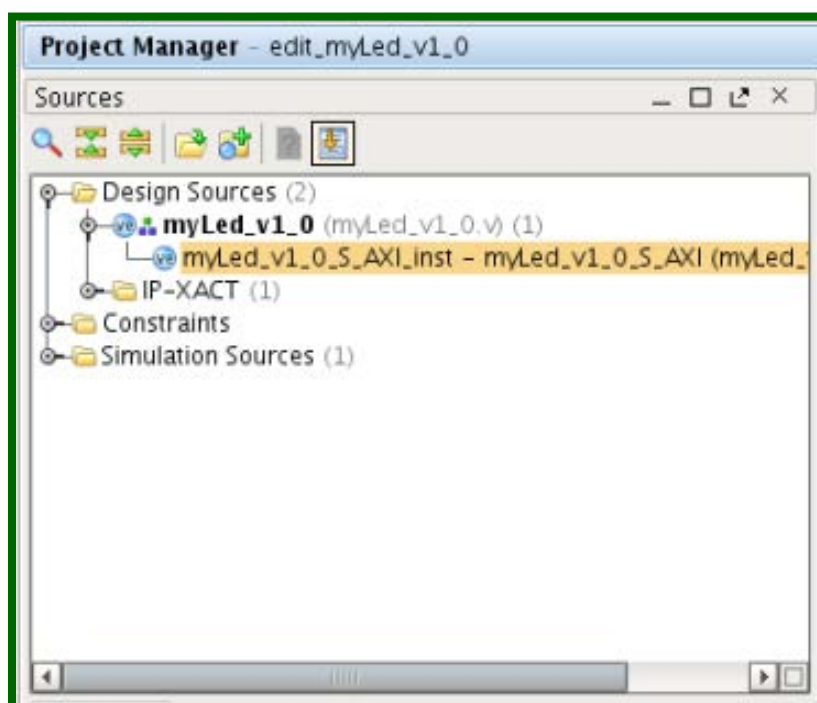


Figure 16. Select user logic file.

```

1  `timescale 1 ps / 1 ps
2
3
4  module myLed_v1_0_S00_AXI #
5  (
6      // Users to add parameters here
7
8      // User parameters ends
9      // Do not modify the parameters beyond this line
10
11     // Width of S_AXI data bus
12     parameter integer C_S_AXI_DATA_WIDTH  = 32,
13     // Width of S_AXI address bus
14     parameter integer C_S_AXI_ADDR_WIDTH  = 4
15 )
16 (
17     // Users to add ports here
18     output wire [3:0] led,
19     // User ports ends
20     // Do not modify the ports beyond this line
21

```

Figure 17. Add user port.

```

395     end
396     end
397 end
398
399 // Add user logic here
400 assign led = slv_reg0[3:0];
401 // User logic ends
402
403 endmodule

```

Figure 18. Add user logic.

10. Next, we need to connect the user logic to myLed. In the project manager select the file **myLed_v_0**. To complete the IP, there are two lines of code we need to add to this file. Under the comment that says “Users to add ports here,” add a port for the LEDs (Fig. 19). Connect the led output from the previous file containing the user logic to myLed (Fig. 20).

```

2  `timescale 1 ps / 1 ps
3
4  module myLed_v1_0 #
5  (
6      // Users to add parameters here
7
8      // User parameters ends
9      // Do not modify the parameters beyond this line
10
11     // Parameters of Axi Slave Bus Interface S00_AXI
12     parameter integer C_S00_AXI_DATA_WIDTH  = 32,
13     parameter integer C_S00_AXI_ADDR_WIDTH  = 4
14 )
15 (
16     // Users to add ports here
17     output [3:0] led,
18     // User ports ends
19

```

Figure 19. Add External.

```

46 // Instantiation of Axi Bus Interface S00_AXI
47 myLed_v1_0_S00_AXI # (
48     .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
49     .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
50 ) myLed_v1_0_S00_AXI_inst (
51     .led(led),
52     .S_AXI_ACLK(s00_axi_aclk),
53     .S_AXI_ARESETN(s00_axi_aresetn),
54     .S_AXI_AWADDR(s00_axi_awaddr),
55     .S_AXI_AWPROT(s00_axi_awprot),
56     .S_AXI_AWVALID(s00_axi_awvalid),
57     .S_AXI_AWREADY(s00_axi_awready),
58     .S_AXI_WDATA(s00_axi_wdata),
59     .S_AXI_WSTRB(s00_axi_wstrb),
60     .S_AXI_WVALID(s00_axi_wvalid),

```

Figure 20. Connect myLed to User Logic.

11. Now that our IP is created and the user logic is defined, we need to package our IP. Under **Project Manager** on the left side of the window, select **Package IP**. A new tab will open that is called Package IP. On the left side of this tap there are a series of labels. We need to complete those that do not have green check marks.

First, select **IP customization Parameters**. At the top of that window select the option to **merge changes from IP Customization Parameters Wizard**, as in Fig. 21.

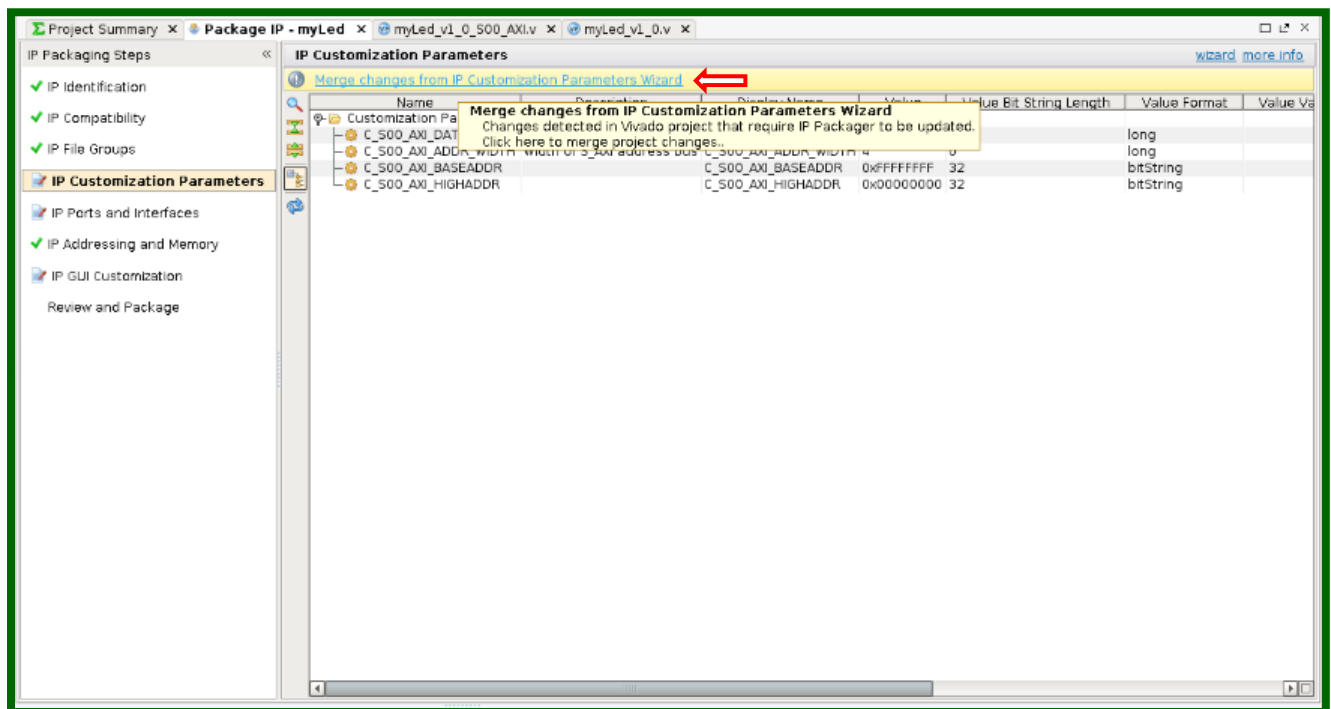


Figure 21. Customization Parameters.

Next, select the IP Ports and Interfaces. Notice that your new LED IP is there (Fig. 22).

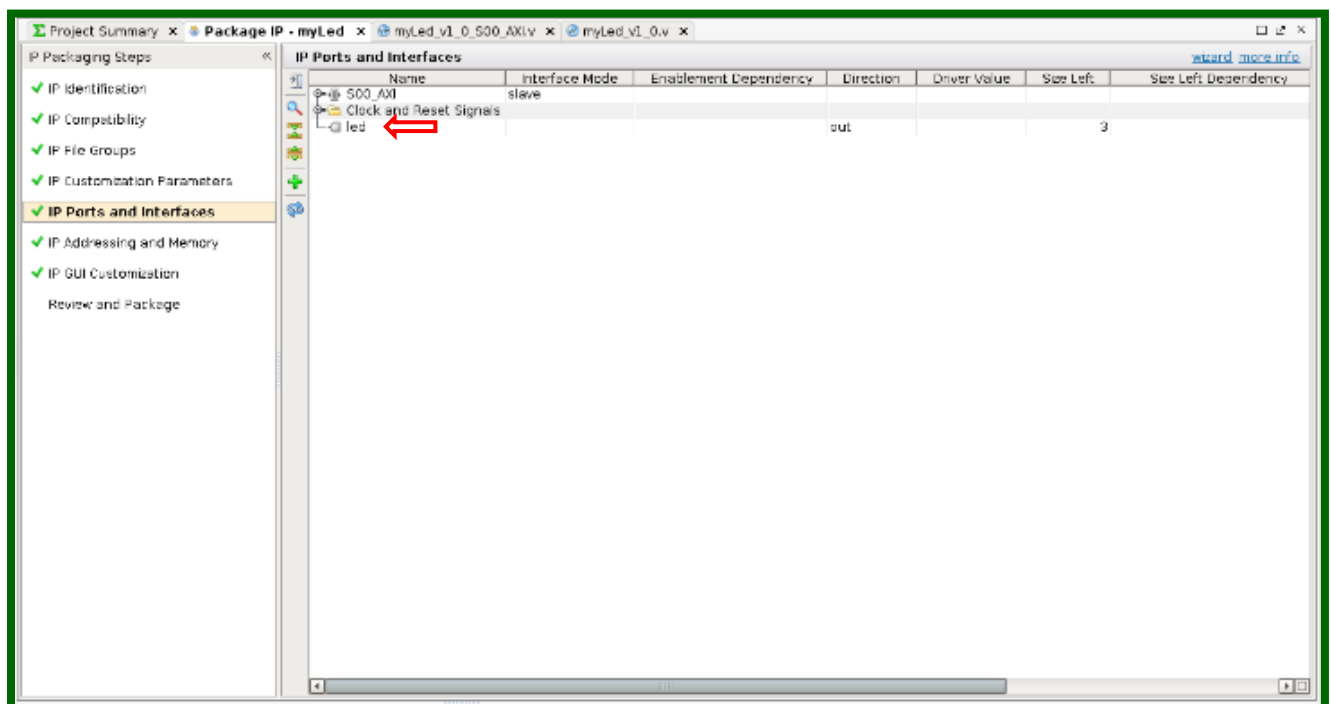


Figure 22. IP Ports and Interfaces.

Next, select **IP GUI Customization**. Our IP GUI is fine as is, so we won't make any changes here (Fig. 23).

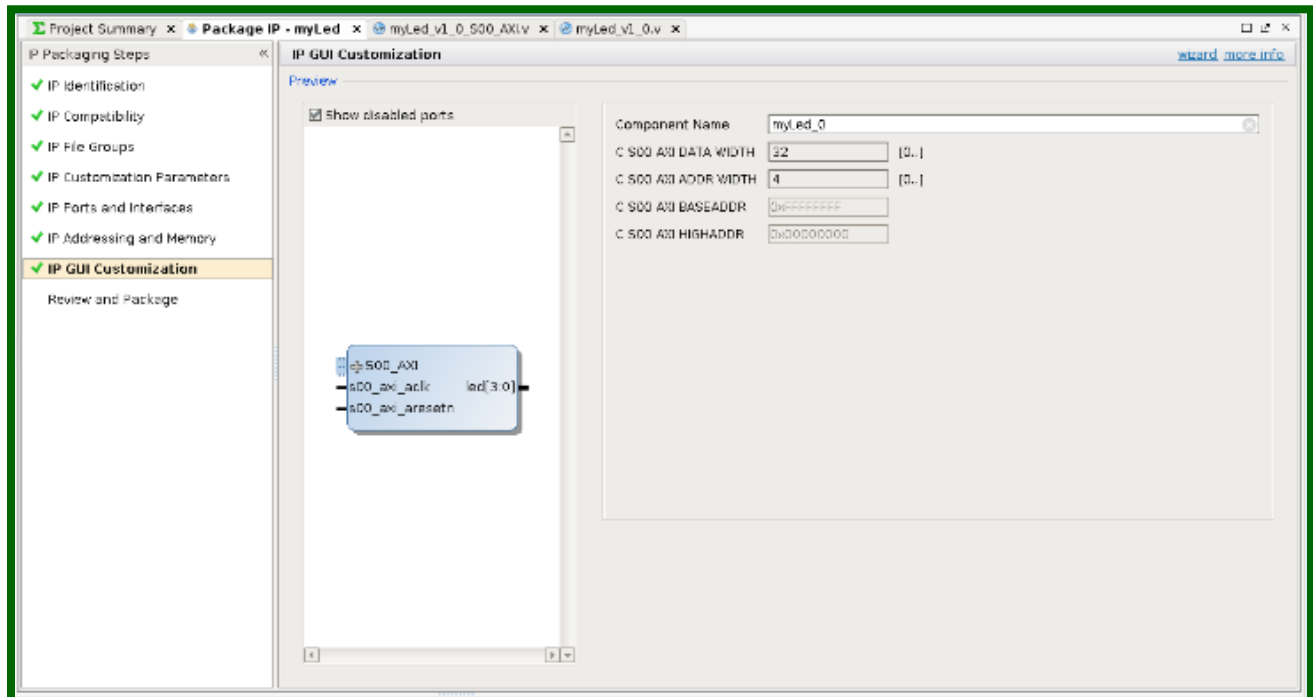


Figure 23. GUI Customization.

Now we can **Review and Package** our myLed IP. Select Review and Package IP and press the **Re-Package IP button**. Our IP is now completed and packaged.

12. We are going to add our IP to our design. Right click anywhere on the block design and click **Add IP** (as shown in Fig. 24). Select the correct IP, **myLed_v1.0**, and press enter (Fig. 23).

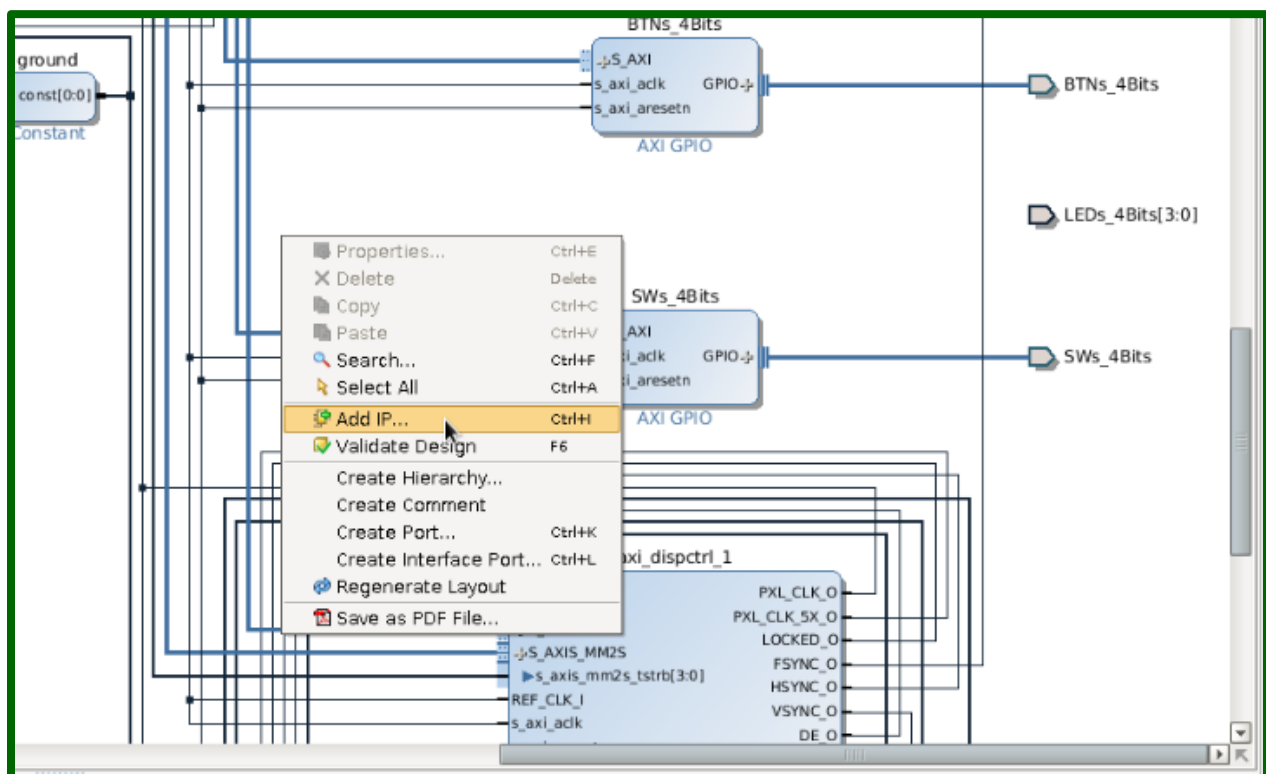


Figure 24. Add IP.

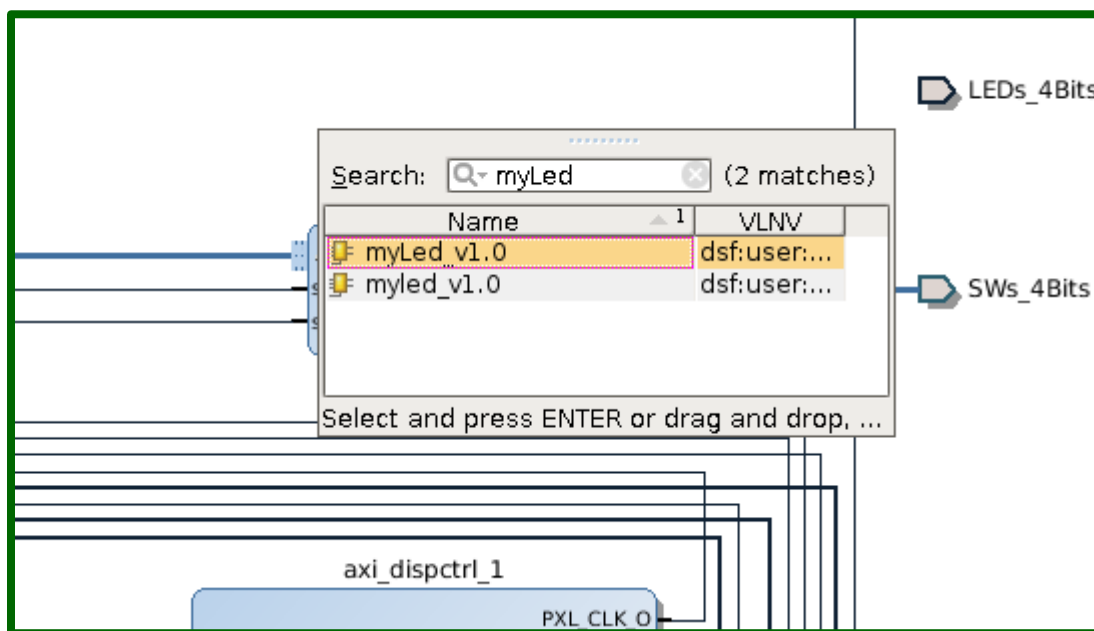


Figure 25. Select IP.

13. The AXI4-Lite bus of myLed IP Core needs to be connected to the processing system. At the top of the window, click the blue text that says **Run connection automation** (Fig. 26). This will connect the inputs of the myLed IP Core. You should see that S_AXI is now connected to the first output of the AXI Interconnect.

If this doesn't pop up go to the address editor
and assign the leds an address

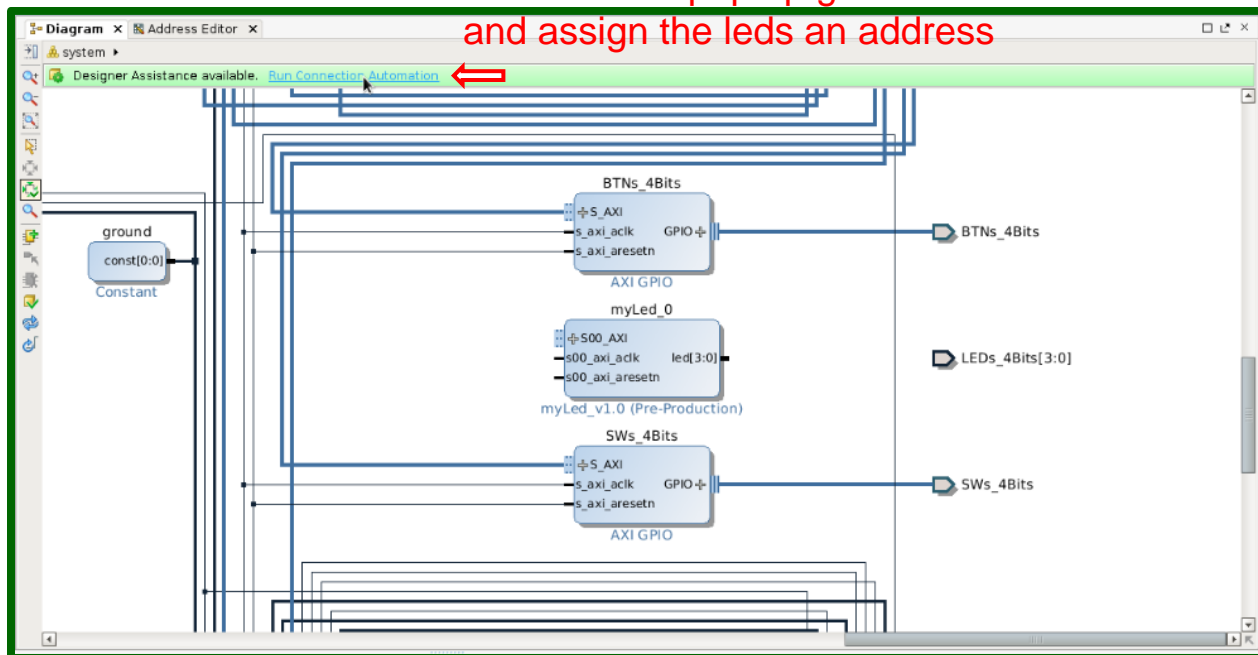


Figure 26. Connect IP.

14. Next, we need to connect the myLed IP to an external port. The myLed IP Core that we implemented will not connect to the existing LEDs_4Bits port, so we need to make a new external port called led. Click on the existing LED port and press delete. To create the new port, right click and select **create port** (Fig. 27). Name the port, select output, select vector [3:0] and press enter.

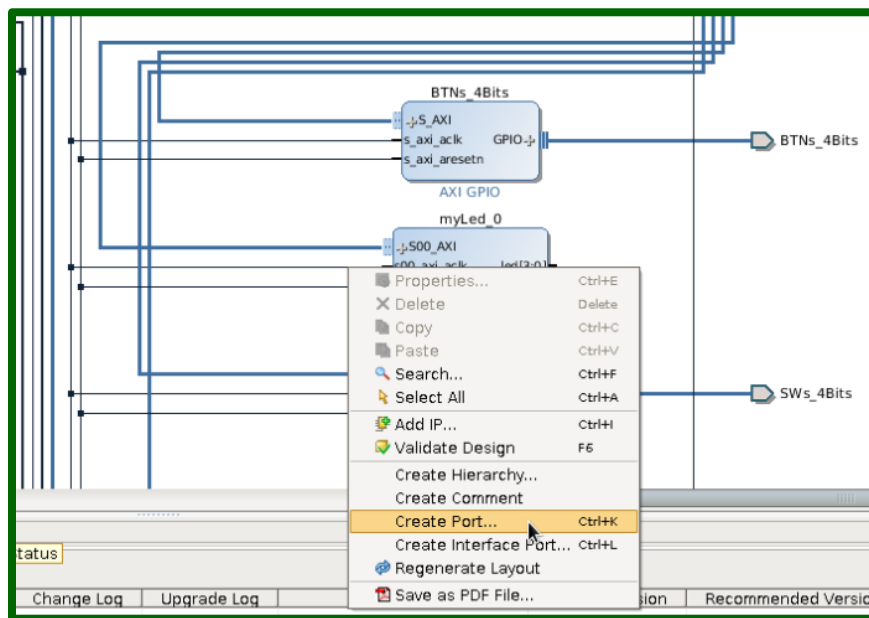


Figure 27. Create Port.

Next, connect the LED port to the myLed IP by clicking on the port and dragging a connection to myLed (Fig. 28).

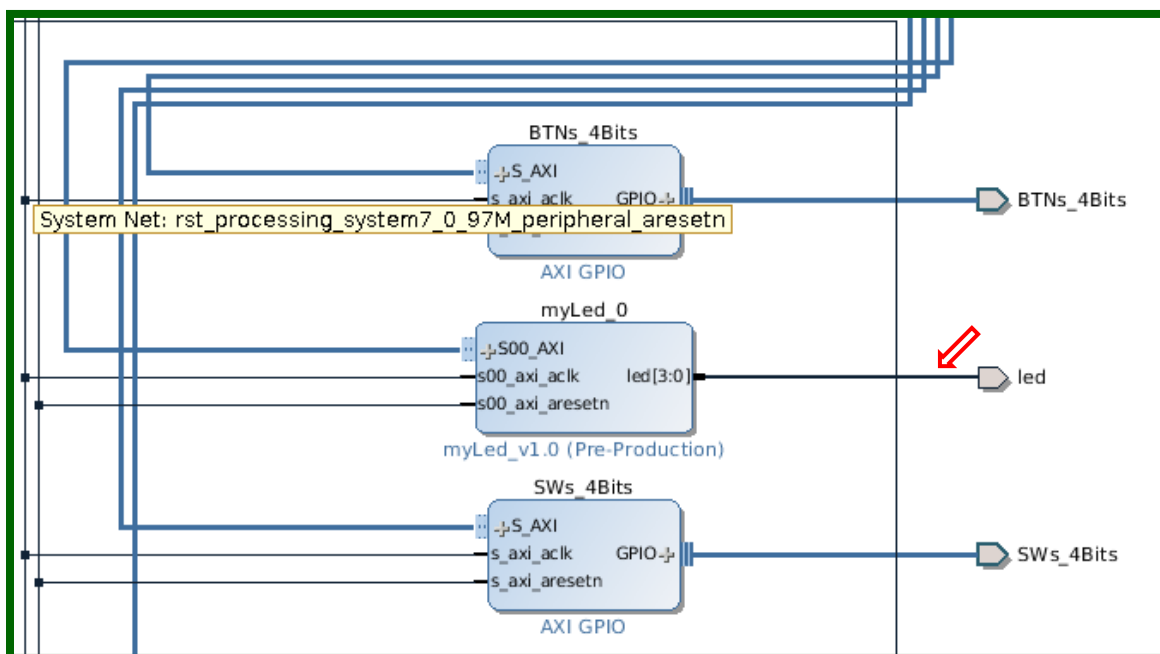


Figure 28. Connect IP to Port.

15. The final step is to specify the pin numbers for myLed_0_LED_pin to physically connect our customized IP Core to the on-board LEDs. In the Project Manager, expand the Constraints section and select the **base.xdc** file (Fig. 29). Within that file, change the names of the external LED pins so that they match the name of our external led port (Fig. 30).

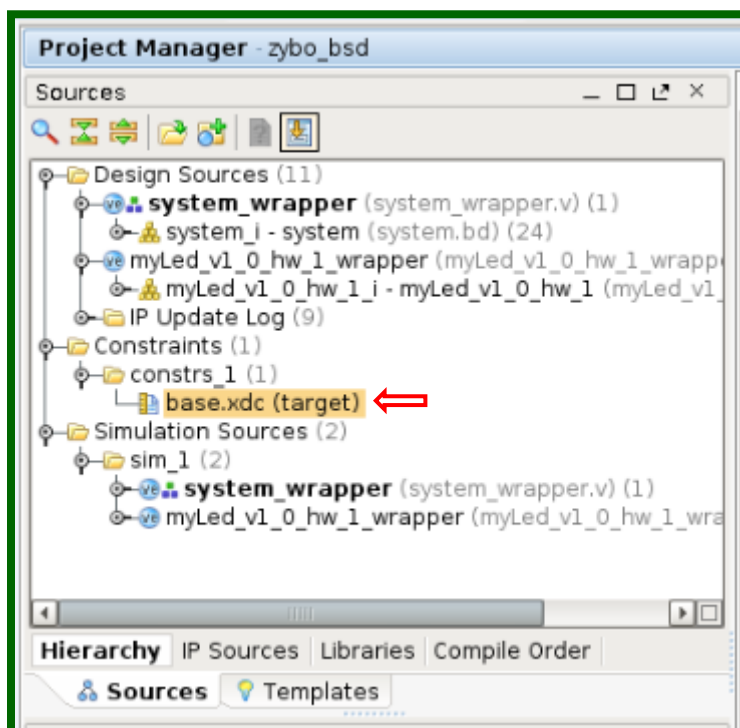


Figure 29. Open XDC File.

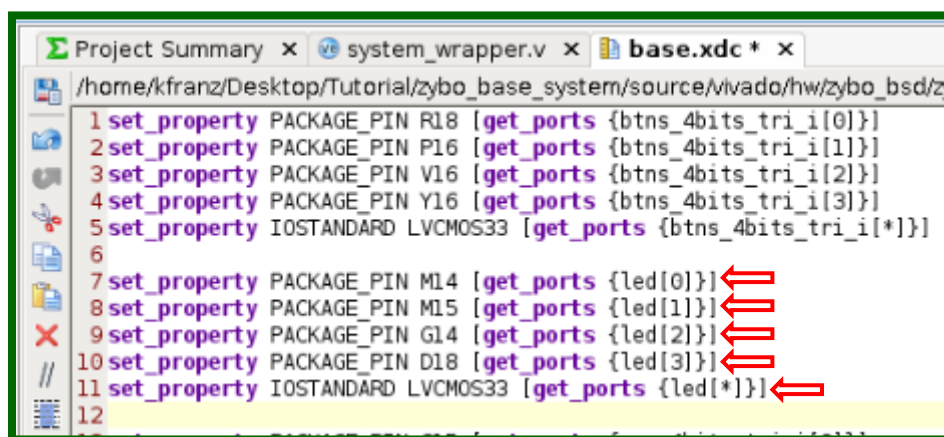


Figure 30. Connect Port led to the LEDs on the ZYBO Board.

16. Regenerate the bitstream for the hardware design by clicking on **Generate Bitstream** under Program and Debug on the left side of the window.

2 Compile U-Boot (this is required and is done with Linux)

2.1 Prerequisites

This is to obtain u-boot.elf file.

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
(Install Xilinx Linux SDK from this site. Get Xilinx.zip which contains Xilinx SDK for Linux. Install the files at /opt/Xilinx under your Linux. Run >source /opt/Xilinx/SDK/2014.4/settings64.sh to setup this SDK.
- ZYBO Base System Design: available at the Digilent Website on the [ZYBO Page](#).


You still need to checkout the master-next branch

2.2 Instructions (Use the Master-Next Branch Until Further Notice)

1. Get the source code for U-Boot from the Digilent Git repository. There are two ways to retrieve the source code:

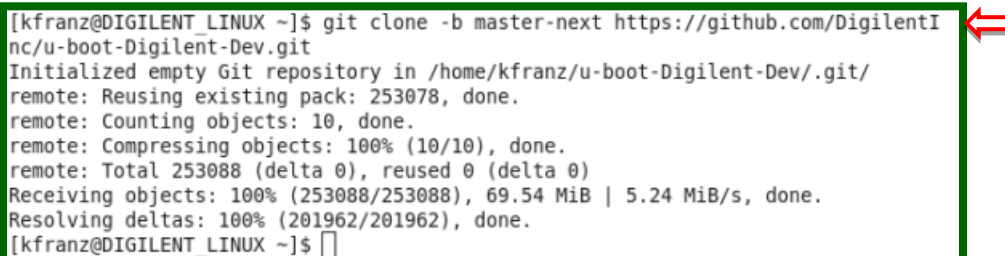
Using *git* command: If you have Git installed in your distribution, you can clone the repository to your computer by command Git clone: <https://github.com/DigilentInc/u-boot-Digilent-Dev.git>. The whole Git Repository is around 55MB, as shown in Fig. 31. If you want to get a separate branch, follow Fig. 32. The next contains the U-boot that is not yet released. The clone URL referenced above can be found on the Digilent GitHub page, as seen in Fig. 33.

Use figure 32 for the command, not figure 31



```
[kfranz@DIGILENT_LINUX ~]$ git clone https://github.com/DigilentInc/u-boot-Digilent-Dev.git
Initialized empty Git repository in /home/kfranz/u-boot-Digilent-Dev/.git/
remote: Reusing existing pack: 253078, done.
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 253088 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (253088/253088), 69.54 MiB | 4.88 MiB/s, done.
Resolving deltas: 100% (201962/201962), done.
```

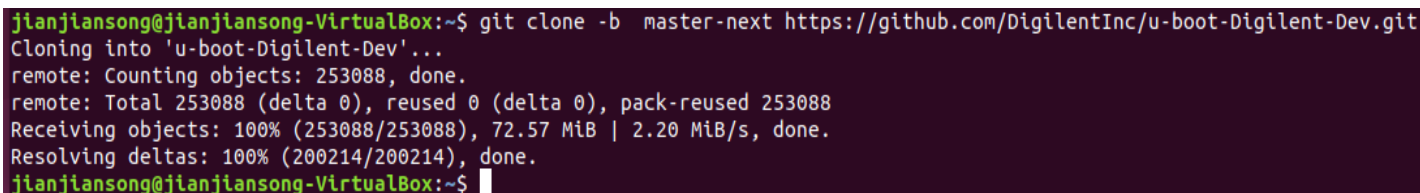
Figure 31. U-Boot repository.



```
[kfranz@DIGILENT_LINUX ~]$ git clone -b master-next https://github.com/DigilentInc/u-boot-Digilent-Dev.git
Initialized empty Git repository in /home/kfranz/u-boot-Digilent-Dev/.git/
remote: Reusing existing pack: 253078, done.
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 253088 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (253088/253088), 69.54 MiB | 5.24 MiB/s, done.
Resolving deltas: 100% (201962/201962), done.
[kfranz@DIGILENT_LINUX ~]$
```

Figure 32. Next-repository.

There is no space between master-next. You can type PgUp and PgDn arrows to recall a command.



```
jianjiansong@jianjiansong-VirtualBox:~$ git clone -b master-next https://github.com/DigilentInc/u-boot-Digilent-Dev.git
Cloning into 'u-boot-Digilent-Dev'...
remote: Counting objects: 253088, done.
remote: Total 253088 (delta 0), reused 0 (delta 0), pack-reused 253088
Receiving objects: 100% (253088/253088), 72.57 MiB | 2.20 MiB/s, done.
Resolving deltas: 100% (200214/200214), done.
jianjiansong@jianjiansong-VirtualBox:~$
```



```
-e 0 \
-n "U-Boot 2014.01-00005-gc29bed918 for zynq board" \
-d u-boot.bin u-boot.img
Image Name: U-Boot 2014.01-00005-gc29bed918
Created: Mon Jan 29 17:47:30 2018
Image Type: ARM U-Boot Firmware (uncompressed)
Data Size: 334528 Bytes = 326.69 kB = 0.32 MB
Load Address: 04000000
Entry Point: 00000000
tools/zynq-boot-bin.py -o boot.bin -u spl/u-boot-spl.bin
/usr/bin/env: 'python': No such file or directory
Makefile:516: recipe for target 'boot.bin' failed
make: *** [boot.bin] Error 127
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$
```

The above error 127 does not matter. Only u-boot is needed.

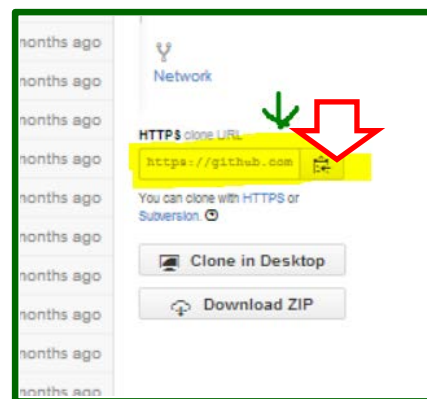


Figure 33. Clone URL.

```
jianjiansong@jianjiansong-VirtualBox:~$ cd u-boot-Digilent-Dev/
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_zybo_config
Configuring for zynq_zybo board...
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

- To compile U-Boot, we need cross-compile tools which are provided by Vivado 2014.1. Those tools have a prefix arm-xilinx-linux-gnueabi- to the standard names for the **GCC** tool chain. The prefix references the platforms that are used. The ZYBO board has two arm cores, so we reference arm. In order to use the cross-platform compilers, please make sure the Vivado 2014.1 settings have been sourced. If not, please refer to step 1 above. To configure and build U-Boot for ZYBO, follow Fig. 34.

ZYBO should be all lowercase. There is a space or two spaces, if one does not work, between gnueabi- and zynq_zybo_config

```
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-
zynq_zybo_config
Configuring for zynq_ZYBO board...
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-
Generating include/autoconf.mk
Generating include/autoconf.mk.dep
...
arm-xilinx-linux-gnueabi-ld -gc-sections -Ttext 0x1000000 -o demo crt0.o demo.o
libgenwrap.o lent-Dev/arch/arm/lib/eabi_compat.o -L /opt/Xilinx/SDK/2014.1/gnu/arm/lin/
bin/./lib/gcc/arm arm-xilinx-linux-gnueabi-objcopy -O binary demo demo.bin 2>/dev/null
make[1]: Leaving directory /home/kfranz/Tutorial/u-boot-Digilent-Dev/examples/api'
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$
```

Figure 34. Compile U-Boot *specify zybo configuration*

- After the compilation, the **ELF** (Executable and Linkable File) generated is named u-boot. We need to add a .elf extension to the file name so that Xilinx SDK can read the file layout and generate BOOT.BIN. In this tutorial, we are going to move the u-boot.elf to the sd_image folder and substitute the u-boot.elf that comes along with the *ZYBO Base System Design Package*, as shown in Fig. 35.

If your are using Windows move the u-boot.elf file to Windows, you still need to change the name though from u-boot to u-boot.elf. copy it to flash drive.

```
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ cp u-boot ../zybo_base_system/sd_image/u-boot.elf
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$
```

Figure 35. Add .elf.

```
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$ cp u-boot u-boot.elf
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$ mv u-boot.elf ~/Desktop/
jianjiansong@jianjiansong-VirtualBox:~/u-boot-Digilent-Dev$
```

3 Generate BOOT.BIN

If Vivado under Windows is used to generate BOOT.BIN. Move the u-boot.elf file to your Windows drive.

3.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
- ZYBO Base System Design: available at the Digilent Website on the [ZYBO Page](#).
- Finished the hardware customization from [Section 1](#) and u-boot .elf from [Section 2](#) (Section 2 optional).

3.2 Instructions

1. Export the hardware design (after Section 1, step 16) to Xilinx SDK by clicking on **File -> Export -> Export Hardware for SDK...**, as shown in Fig. 36.

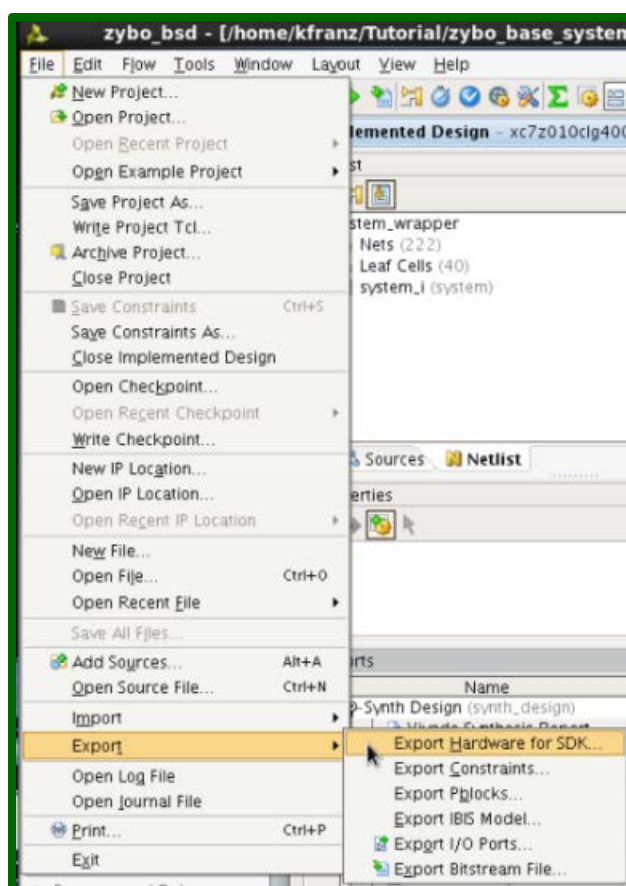


Figure 36. Export Hardware Design to SDK.

2. Leave the workspace as **<Local to Project>**. Make sure that the “Launch SDK” box is checked and click **OK**, as shown in Fig. 37.

Note: If you are using Vivado 2014.1, you may have to export twice.

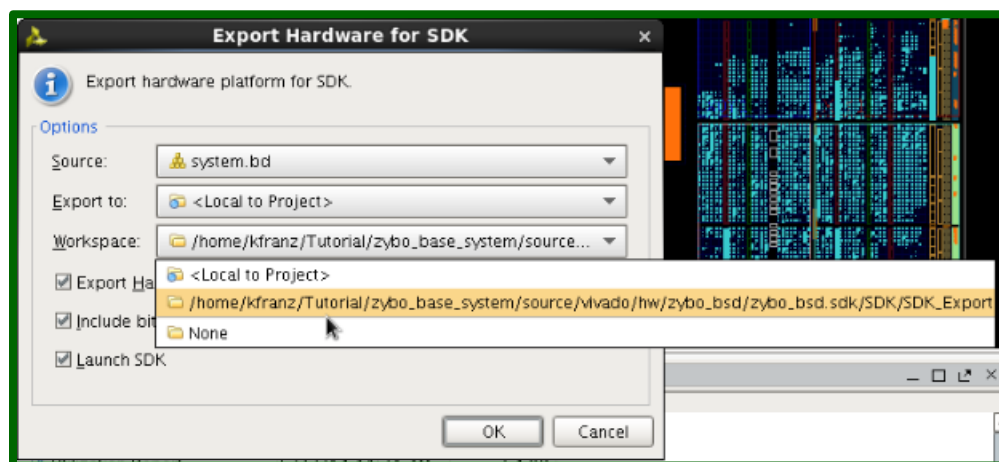


Figure 37. Set SDK Workspace Path.

- After SDK launches, the hardware platform project is already present in Project Explorer on the left of the SDK main window, as shown in Fig. 38. We now need to create a First Stage Bootloader (FSBL). Click **File->New->Project...**, as shown in Fig. 39.

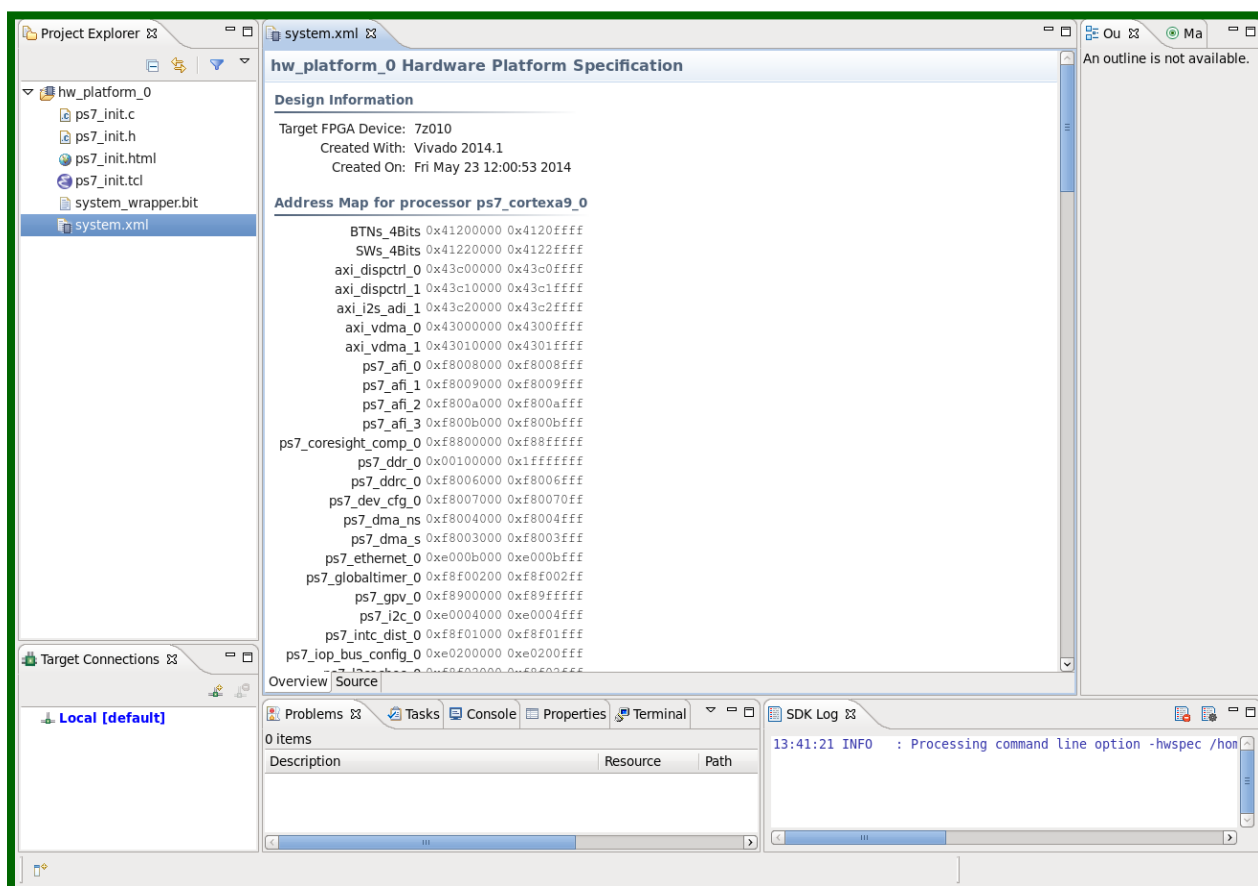


Figure 38. Export hardware design to SDK.

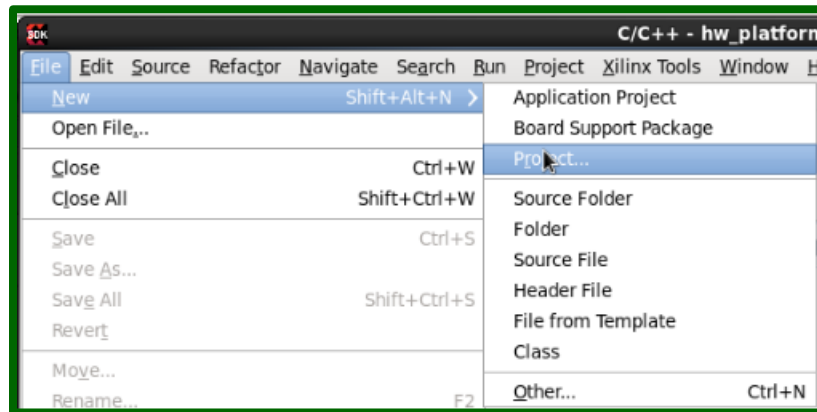


Figure 39. Create new project in SDK.

4. In the New Project window, select **Xilinx->Application Project**, and then Click **Next** (Fig. 40).

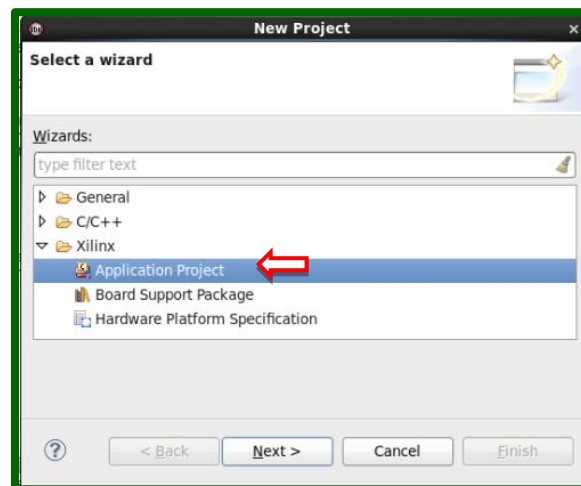


Figure 40. Select Application Project Wizard.

5. We will name the project FSBL. Select **hw_platform_0** for **Target Hardware** because it is the hardware project we just exported. Select **standalone** for **OS Platform**. Select **use existing board support package**. Click **Next**, as shown in Fig. 41.

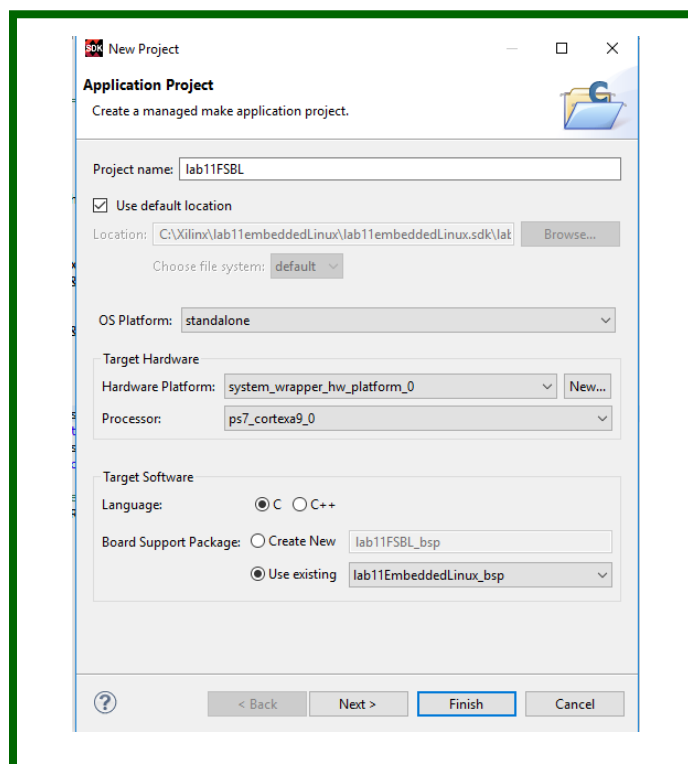


Figure 41. New Application Project.

6. Select **Zynq FSBL** as template, and click Finish as shown in Fig. 42.

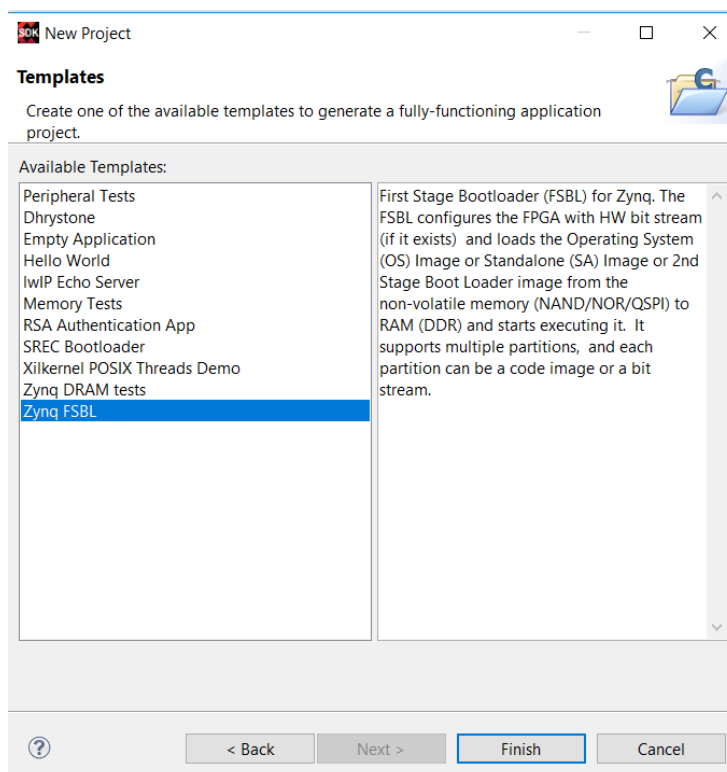


Figure 42. Select Zynq FSBL as template.

7. For the ZYBO, we need to set the mac address for the Ethernet in the fsbl hook. We want the mac address for the Ethernet to remain constant when we turn the ZYBO board off and on. You can swap the fsbl_hooks.c file in the FSBL project with the fsbl_hooks.c under source/vivado/SDK/fsbl in the ZYBO Base System Design (Fig. 43).

Delete fsbl_hooks.c and import fsbl_hooks.c from Zybo Base System Design folder.

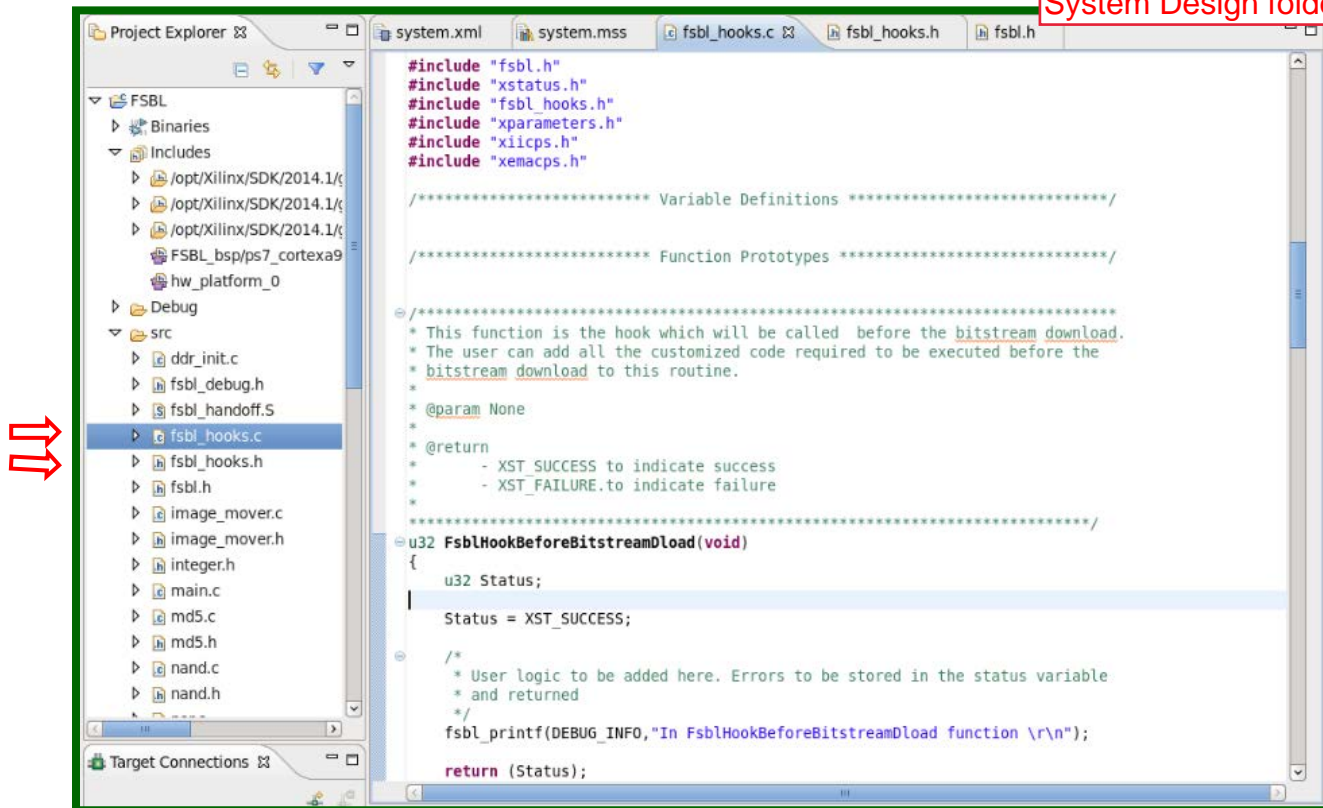


Figure 43. fsbl_hooks.c

8. After you have saved the changes to fsbl_hooks.c, the project will rebuild itself automatically. If it does not rebuild, click **Project->Clean** to clean the project files, and **Project->Build All** to rebuild all the projects. The compiled ELF file is located in:

Your FSBL project folder under: /Debug

ZYBO_base_system/source/vivado/hw/ZYBO_bsd.sdk/SDK/SDK_Export/FSBL/Debug

9. Now we have all of the files ready to create BOOT.BIN. Click **Xilinx Tools -> Create Zynq Boot Image**, as shown in Fig. 44.

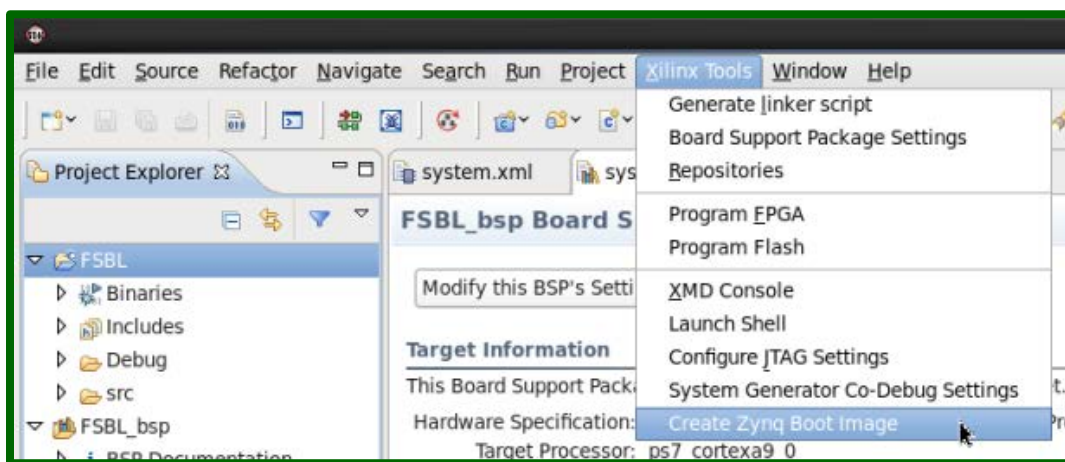


Figure 44. Create Zynq Boot Image.

10. In the Create Zynq Boot Image window (as shown in Fig. 45), Click **Browse** to set the path for **FSBL elf**.
 Click **Add** to add the **system.bit** file found at: **It's system_wrapper.bit under *.sdk->platform**
 /ZYBO_base_system/source/vivado/hw/ZYBO_bsd/ZYBO_bsd.sdk/SDK/SDK_Export
 /hw_platform_0/. Click **Add** to add the **u-boot.elf** file found at:
 ZYBO_base_system/sd_image/. **It is very important that the 3 files are added in this order, or else**
 the FSBL will not work properly (the proper order can be seen in Fig. 45). It is also very important that you
 set FSBL.elf as the bootloader and system.bit and u-boot.elf as data files. In this tutorial, the
 sd_image folder is set as output folder for the BIN file. Click **Create Image**.

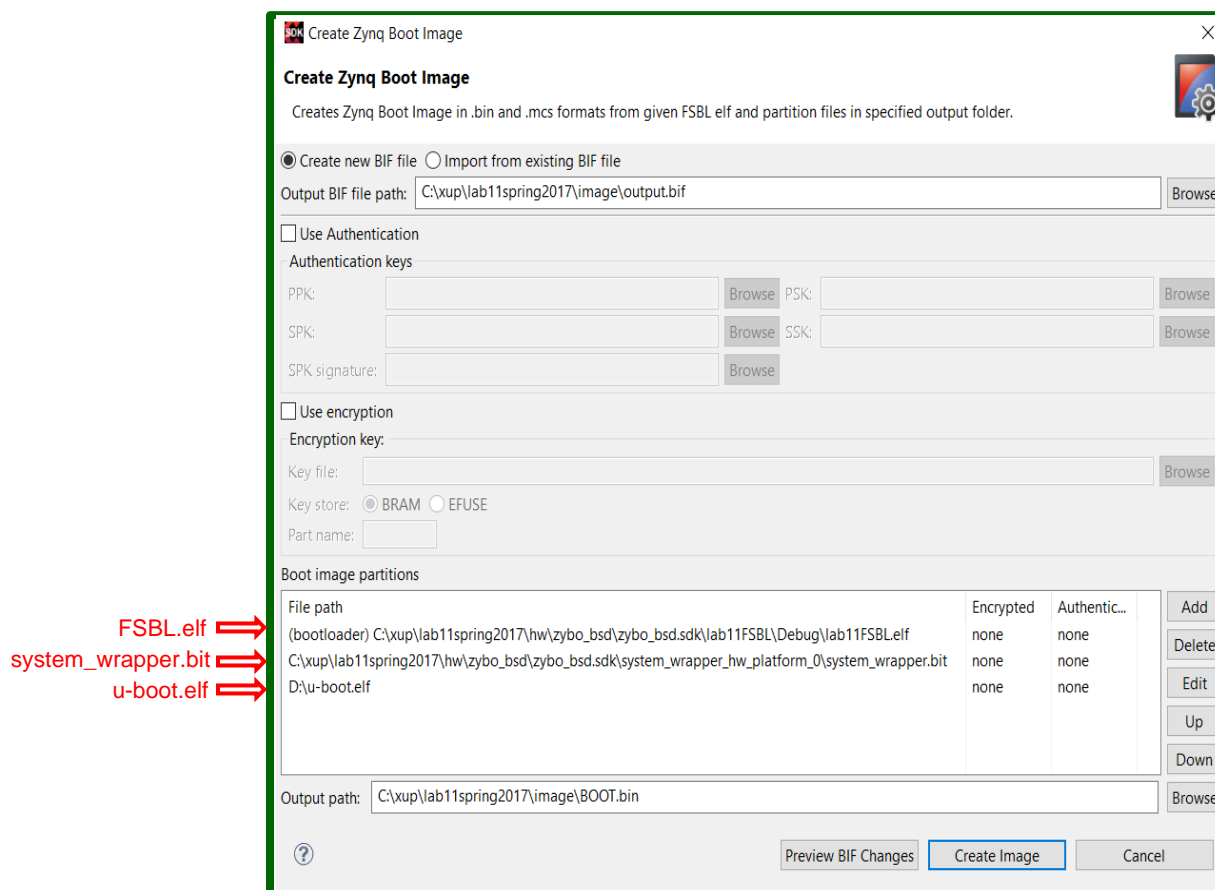


Figure 45. Create Zynq Boot Image Configuration.

11. The created BIN file was named BOOT.bin.

This is to obtain ulmage binary file as Linux kernel, stored in Linux-Digilent-Dev/arch/arm/boot/

4 Compile Linux Kernel

Commands are executed under Linux-Digilent-Dev/

4.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
- ZYBO Base System Design: available at the Digilent Website on the [ZYBO Page](#).

You still need to checkout master-next

4.2 Instructions (Use the Master-Next Branch Until Further Notice)

1. Get the Linux kernel source code from Digilent Git repository. There are two ways to retrieve the source code: The correct command is: `git clone -b master-next https://github.com/DigilentInc/Linux-Digilent-Dev.git`

Using git command: If you have Git installed in your distribution, you can clone the repository to your computer by command `git clone https://github.com/DigilentInc/Linux-Digilent-Dev.git`
 The whole Git Repository is around 850MB, as shown in Fig. 46.
The only folder that is needed is Linux-Digilent-Dev-arch-arm.

```
jlanjlansong@jlanjlansong-VirtualBox:~$ git clone -b master-next https://github.com/DigilentInc/Linux-Digilent-Dev.git
Cloning into 'Linux-Digilent-Dev'...
remote: Counting objects: 3902315, done.
remote: Total 3902315 (delta 0), reused 0 (delta 0), pack-reused 3902315
Receiving objects: 100% (3902315/3902315), 946.20 MiB | 2.27 MiB/s, done.
Resolving deltas: 100% (3267882/3267882), done.
Checking out files: 100% (48202/48202), done.
jlanjlansong@jlanjlansong-VirtualBox:~$
```

Figure 46. Clone Kernel.

You may need to close your VirtualBox and start it again to update all environment parameters. If you get any error when you execute the following commands, restart your VirtualBox.

located at arch/arm/configs/xilinx_zynq_defconfig. To use the default configuration, you can follow Fig. 47. Change directory to /Linux-Digilent-Dev.

Run the command in figure 50 first to add u-boot tools path.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- xilinx_zynq_defconfig
#
# configuration written to .config
#
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

3. Follow Fig. 48 to compile the Linux Kernel.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- scripts/kconfig/conf --silentoldconfig Kconfig
CHK include/config/kernel.release
CHK include/generated/uapi/linux/version.h
CHK include/generated/utsrelease.h
```

```
jlanjlansong@jlanjlansong-VirtualBox:~/Linux-Digilent-Dev$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- xilinx_zynq_defconfig
#
# configuration written to .config
#
jlanjlansong@jlanjlansong-VirtualBox:~/Linux-Digilent-Dev$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- scripts/kconfig/conf --silentoldconfig Kconfig
jlanjlansong@jlanjlansong-VirtualBox:~/Linux-Digilent-Dev$
```

The command is: `make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- scripts/kconfig/conf --silentoldconfig Kconfig` It is not necessary to generate zImage in order to generate uImage. This command may take more than ten minutes.


```
Kernel: arch/arm/boot/Image is ready
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
Building modules, stage 2.
MODPOST 23 modules
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

If mkimage command is not found, run:
sudo apt install u-boot-tools

Figure 48. Compile Kernel.

Run the command in figure 50 first, change it to use your username and path
 Install u-boot-tools: **sudo apt install u-boot-tools** before next command.

- After the compilation, the kernel image is located at arch/arm/boot/zImage. However, in this case the kernel image has to be a ulmage (unzipped) rather than a zimage. To make the uimage, follow Fig. 49.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- UIMAGE_LOADADDR=0x8000 uImage
CHK     include/config/kernel.release
CHK     include/generated/uapi/linux/version.h
CHK     include/generated/utsrelease.h
make[1]: `include/generated/mach-types.h' is up to date.
```

This may take about 17 minutes.

```
Data Size: 3819232 Bytes = 3729.72 kB = 3.64 MB
Load Address: 00000000
Entry Point: 00000000
Image arch/arm/boot/uImage is ready
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

Figure 49. Create ulmage. *ulmage can be generated without zImage.*

Note: Depending on your distribution of Linux, you may get an error regarding the path of the mkimage. If this is the case, you can change the path following Fig. 50.

```
[kfranz@DIGILENT_LINUX Tutorial]$ PATH=$PATH:/home/kfranz/Tutorial/u-boot-Digilent-Dev/tools/
[kfranz@DIGILENT_LINUX Tutorial]$ echo $PATH
```

Figure 50. Change Path.

```
checking out latest 100% (10262/10262): done!
jianjiansong@jianjiansong-VirtualBox:~$ ls
core Desktop Documents Downloads examples.desktop Linux-Digilent-Dev Music Pictures Public Templates u-boot-Digilent-Dev Videos
jianjiansong@jianjiansong-VirtualBox:~$ PATH=$PATH:/home/jianjiansong/u-boot-Digilent-Dev/tools/
jianjiansong@jianjiansong-VirtualBox:~$ echo $PATH
/opt/Xilinx/SDK/2015.2/bin:/opt/Xilinx/SDK/2015.2/gnu/microblaze/lin/bin:/opt/Xilinx/SDK/2015.2/gnu/arm/lin/bin:/opt/Xilinx/SDK/2015.2/gnu/microblaze/linux_toolchain/lin64_be/bin:/opt/Xilinx/SDK/2015.2/gnu/microblaze/linux_toolchain/lin64_le/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/jianjiansong/u-boot-Digilent-Dev/tools/
jianjiansong@jianjiansong-VirtualBox:~$
```

5 Test Kernel Image with Pre-built File System

5.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
- Linux Kernel Source Code: available at Digilent GitHub repository <https://github.com/DigilentInc/Linux-Digilent-Dev>. (Use the Master-Next Branch Until Further Notice)
- Pre-built File System Image: ramdisk Image is available in ZYBO Linux Reference Design.
- BOOT.BIN from **Section 3**, ulmage from **Section 4**.

5.2 Instructions **make uramdisk.image.gz and devicetree.dtb**

- To boot the Linux operating system on the ZYBO, you need BOOT.BIN, a Linux kernel image (ulmage), a device tree blob (DTB file), and a file system. BOOT.BIN has been created in Section 3 and ulmage has been compiled in Section 4. We will now compile the DTB file. The default device tree source file is located in the Linux Kernel source at arch/arm/boot/dts/zynq-ZYBO.dts.

RAMDISK: modify the device tree source file according to Fig. 51. For Zynq, only the ramdisk image has to be wrapped in a u-boot header in order for u-boot to boot with it. This is shown in Fig. 52.

Skip this step in Figure 51, DO NOT change the zynq_zybo.dts file

```

49      /* bootargs = "console=ttyPS0,115200 root=/dev/mmcbk0p2 rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=1"; */
50      bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M
init=/init earlyprintk rootwait devtmpfs.mount=1";
51      linux,stdout-path = "/axi0/serial@e0001000";
52  };

```

Figure 51. Edit device tree.

uramdisk.image.gz

.Download arm_ramdisk.image.gz from <http://www.wiki.xilinx.com->Build+and+Modify+a+Rootfs>. Store it under user home directory. Run the following command instead of Figure 52.

.u-boot-Digilent-Dev/tools/mkimage -A arm -T ramdisk -c gzip -d ./arm_ramdisk.image.gz uramdisk.image.gz

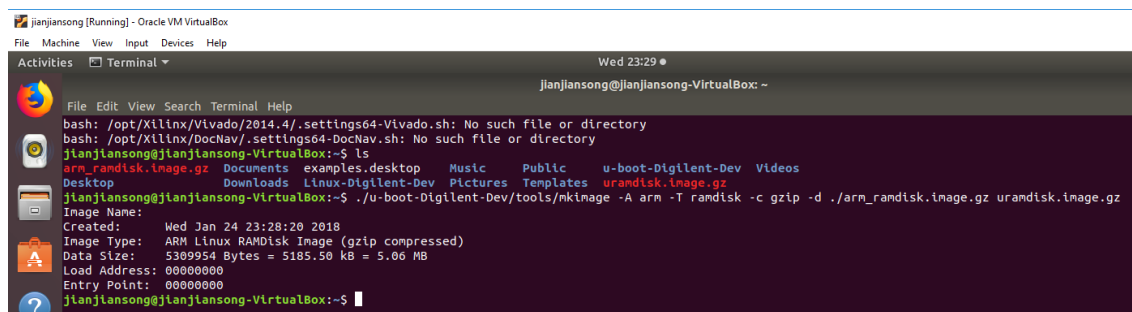


Figure 52. Make Uramdisk Image.

make devicetree.dtb 2. Generate DTB file, as shown in Fig. 53.

```

[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ ./scripts/dtc/dtc -I dts -O dtb -o .
./devicetree.dtb arch/arm/boot/dts/zynq-zybo.dts
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$

```



There is no space between the two dots.

sudo apt install device-tree-compiler
install dtc

Figure 53. Generate DTB File.

-O and -o are upper and lower case letter O
devicetree.dtb is stored in home directory.

3. **(RAMDISK)** Copy BOOT.BIN, devicetree.dtb, ulmage and uramdisk.image.gz to the first partition of an SD card, as shown in Fig. 54.

Copy four files to an SD card that has the format FAT32: BOOT.BIN, devicetree.dtb, ulmage, uramdisk.iamge.gz

```

[kfranz@DIGILENT_LINUX Tutorial]$ ls
devicetree.dtb linux-digilent-dev u-boot-digilent ZYBO_base_system
[kfranz@DIGILENT_LINUX Tutorial]$ cp ZYBO_base_system/sd_image/BOOT.BIN /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp ZYBO_base_system/sd_image/ uramdisk.image.gz /BOOT.BIN
/media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp ./devicetree.dtb /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp Linux-Digilent-Dev/arch/arm/boot/uImage /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$

```

Figure 54. Ramdisk.

Jumper 7 doesn't exist

4. Plug the SD card into the ZYBO. To boot from the SD card, jumper 7 needs to be configured for USB, as shown on the ZYBO board, and Jumper 5 must be connected to SD. Connect UART port to PC with a micro USB cable and set the UART terminal on PC to 115200 baud rate, 8 data bits, 1 stop bit, no parity, and no flow control. After powering on the board, the console (shown in Fig. 55) should be seen at the UART terminal if you use RamDisk. More information about these file systems can be found in *Getting Started with Embedded Linux - ZYBO*.

```

[ 1.170000] fb0: frame buffer device
[ 1.170000] drm: registered panic notifier
[ 1.170000] [drm] Initialized analog_drm 1.0.0 20110530 on minor 0
[ 1.260000] EXT4-fs (ram0): couldn't mount as ext3 due to feature incompatibilities
[ 1.310000] EXT4-fs (ram0): mounting ext2 file system using the ext4 subsystem
[ 1.310000] EXT4-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
[ 1.320000] EXT4-fs (ram0): mounted filesystem without journal. Opts: (null)
[ 1.320000] UFS: Mounted root (ext2 filesystem) on device 1:0.
[ 1.330000] devtmpfs: mounted
[ 1.330000] Freeing init memory: 152K
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Configure static IP 192.168.1.10
[ 1.510000] GEM: lp->tx_bd ffdfa000 lp->tx_bd_dma 19bd7000 lp->tx_skb d8070480
[ 1.520000] GEM: lp->rx_bd ffdfb000 lp->rx_bd_dma 19bd8000 lp->rx_skb d8070580
[ 1.520000] GEM: MAC 0x00350a00. 0x00002201, 00:0a:35:00:01:22
[ 1.530000] GEM: phydev d8b6f400, phydev->phy_id 0x1410dd1, phydev->addr 0x0
[ 1.530000] eth0, phy_addr 0x0, phy_id 0x01410dd1
[ 1.540000] eth0, attach [Marvell 88E1510] phy driver
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
++ Starting OLED Display
[ 1.580000] pmodoled-gpio-spi [zed_oled] SPI Probing
++ Exporting LEDs & SWs
rcS Complete
zynq>

```

Figure 55. Ramdisk, UART Console after boot up.

6 Modify Device Tree and Compose Kernel Driver

6.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
- Linux Kernel Source Code: available at Diligent GitHub repository <https://github.com/Diligentinc/Linux-Diligent-Dev> (Use the Master-Next Branch Until Further Notice)

6.2 Instructions

1. Create a directory named “drivers” in the Tutorial folder, as shown in Fig. 56. Inside the driver’s directory, we will compose the driver for the myLed controller.

```

[kfranz@DIGILENT_LINUX Tutorial]$ mkdir drivers
[kfranz@DIGILENT_LINUX Tutorial]$ ls
BOOT.bin          output.bif          vivado.jou
drivers           ps_clock_registers.log  vivado.log
Linux-Diligent-Dev u-boot-Diligent-Dev  zybo_base_system
[kfranz@DIGILENT_LINUX Tutorial]$

```

Figure 56. Driver Directory.

2. We need a Makefile so that we can compile the driver. The Makefile is created in Fig. 57.

```

jianjiansong@jianjiansong-VirtualBox:~$ cd drivers/
jianjiansong@jianjiansong-VirtualBox:~/drivers$ gedit Makefile
jianjiansong@jianjiansong-VirtualBox:~/drivers$

```

Figure 57. Create Makefile.

Use nano or gedit. Vim may mess up tabs sometimes.

After creating the file, hit I to change to insert mode and insert the following text (Fig. 58).

all: and clean: should be all the way to the left and the make. commands should be tabbed once.

```
obj-m := myled.o

all:
    make -C ../Linux-Digilent-Dev/ M=$(PWD) modules

clean:
    make -C ../Linux-Digilent-Dev/ M=$(PWD) clean
```

Figure 58. Makefile.

Pay attention to this note:

Note: make sure the spacing in the Makefile is made up of tabs, not spaces, where necessary. Hit control^s to save the file and click x icon to exit gedit.

3. We will start with a simple driver that creates a file named myled under the Linux /proc file system. The status of the on-board LEDs can be changed by writing a number to the file. The driver is coded in Fig. 59.

```
kfranz@DIGILENT LINUX drivers]$ vim myled.c
```

Figure 59. Create myled.c

If you see error: xilinx-linux-gnueabi-gcc: not found, you will need to install 64-bit library using:

`sudo apt-get install gcc-arm-none-eabi gcc-arm-linux-gnueabi`

You may also need to install some 32-bit libraries. Check Internet to find instructions.

```
jianjiansong@jianjiansong-VirtualBox:~/drivers$ ls
devicetree.dtb Makefile myled.c zynq-zybo.dts
jianjiansong@jianjiansong-VirtualBox:~/drivers$ gedit Makefile
jianjiansong@jianjiansong-VirtualBox:~/drivers$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make -C ../Linux-Digilent-Dev/ M=/home/jianjiansong/drivers modules
make[1]: Entering directory '/home/jianjiansong/Linux-Digilent-Dev'
CC [M] /home/jianjiansong/drivers/myled.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/jianjiansong/drivers/myled.mod.o
LD [M] /home/jianjiansong/drivers/myled.ko
make[1]: Leaving directory '/home/jianjiansong/Linux-Digilent-Dev'
jianjiansong@jianjiansong-VirtualBox:~/drivers$ ls
devicetree.dtb modules.order myled.c myled.mod.c myled.o
Makefile Module.symvers myled.ko myled.mod.o zynq-zybo.dts
jianjiansong@jianjiansong-VirtualBox:~/drivers$
```

Re: PC with Ubuntu 14.04 64bit, SDK 2015.1 fails to compile (make error 2) compiler not found

[Options ▾](#)

05-15-2015 02:02 PM

This has always been the case with the toolchain on 64-bit Ubuntu. I used to need these to just start Vivado when it was 32-bit, and now after upgrading to the 64-bit 2015.1, I forgot all about it until my system complained that *arm-xilinx-eabi-gcc could not be found*. These are third-party (GNU) projects/programs. Xilinx just packs a stable version in with their installer for convenience, which they probably should have built for amd64 platforms if they're dropping support for 32-bit otherwise, but they're likely just going to push it out later.

If you'd rather use a 64-bit version, you can install it with apt-get or whatever your distro's package manager is, then in the SDK you can change the project settings to use your own toolchain.

```
sudo apt-get install gcc-arm-none-eabi gcc-arm-linux-gnueabi
```

If you think Xilinx is bad, try using TI's Code Composer Studio and the TI-RTOS/SYSBIOS real-time system... I did, and I've cherished every moment I spend with the Xilinx tools ever since, lol, and TI doesn't have to integrate a whole toolchain for FPGA development either.

```

1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <asm/uaccess.h>      /* Needed for copy_from_user */
4 #include <asm/io.h>          /* Needed for IO Read/Write Functions */
5 #include <linux/proc_fs.h>    /* Needed for Proc File System Functions */
6 #include <linux/seq_file.h>   /* Needed for Sequence File Operations */
7 #include <linux/platform_device.h> /* Needed for Platform Driver Functions */
8
9 /* Define Driver Name */
10 #define DRIVER_NAME "myled"
11
12 unsigned long *base_addr;    /* Virtual Base Address */
13 struct resource *res;        /* Device Resource Structure */
14 unsigned long remap_size;    /* Device Memory Size */
15
16 /* Write operation for /proc/myled
17 * -----
18 * When user cat a string to /proc/myled file, the string will be stored in
19 * const char __user *buf. This function will copy the string from user
20 * space into kernel space, and change it to an unsigned long value.
21 * It will then write the value to the register of myled controller,
22 * and turn on the corresponding LEDs eventually.
23 */
24 static ssize_t proc_myled_write(struct file *file, const char __user * buf,
25                               size_t count, loff_t * ppos)
26 {
27     char myled_phrase[16];
28     u32 myled_value;
29
30     if (count < 11) {
31         if (copy_from_user(myled_phrase, buf, count))
32             return -EFAULT;
33
34         myled_phrase[count] = '\0';
35     }
36
37     myled_value = simple_strtoul(myled_phrase, NULL, 0);
38     wmb();
39     iowrite32(myled_value, base_addr);
40     return count;
41 }
42
43 /* Callback function when opening file /proc/myled
44 * -----
45 * Read the register value of myled controller, print the value to
46 * the sequence file struct seq_file *p. In file open operation for /proc/myled
47 * this callback function will be called first to fill up the seq_file,
48 * and seq_read function will print whatever in seq_file to the terminal.
49 */
50 static int proc_myled_show(struct seq_file *p, void *v)
51 {
52     u32 myled_value;
53     myled_value = ioread32(base_addr);
54     seq_printf(p, "0x%x", myled_value);
55     return 0;
56 }
57

```

Add #include <linux/slab.h> after all the #include statements




Figure 60. myled.c

```

58 /* Open function for /proc/myled
59 * -----
60 * When user want to read /proc/myled (i.e. cat /proc/myled), the open function
61 * will be called first. In the open function, a seq_file will be prepared and the
62 * status of myled will be filled into the seq_file by proc_myled_show function.
63 */
64 static int proc_myled_open(struct inode *inode, struct file *file)
65 {
66     unsigned int size = 16;
67     char *buf;
68     struct seq_file *m;
69     int res;
70
71     buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
72     if (!buf)
73         return -ENOMEM;
74
75     res = single_open(file, proc_myled_show, NULL);
76
77     if (!res) {
78         m = file->private_data;
79         m->buf = buf;
80         m->size = size;
81     } else {
82         kfree(buf);
83     }
84
85     return res;
86 }
87
88 /* File Operations for /proc/myled */
89 static const struct file_operations proc_myled_operations = {
90     .open = proc_myled_open,
91     .read = seq_read,
92     .write = proc_myled_write,
93     .llseek = seq_lseek,
94     .release = single_release
95 };
96
97 /* Shutdown function for myled
98 * -----
99 * Before myled shutdown, turn-off all the leds
100 */
101 static void myled_shutdown(struct platform_device *pdev)
102 {
103     iowrite32(0, base_addr);
104 }
105

```

Figure 60. myled.c (Cont.)


```

106 /* Remove function for myled
107 * -----
108 * When myled module is removed, turn off all the leds first,
109 * release virtual address and the memory region requested.
110 */
111 static int myled_remove(struct platform_device *pdev)
112 {
113     myled_shutdown(pdev);
114
115     /* Remove /proc/myled entry */
116     remove_proc_entry(DRIVER_NAME, NULL);
117
118     /* Release mapped virtual address */
119     iounmap(base_addr);
120
121     /* Release the region */
122     release_mem_region(res->start, remap_size);
123
124     return 0;
125 }
126
127 /* Device Probe function for myled
128 * -----
129 * Get the resource structure from the information in device tree.
130 * request the memory region needed for the controller, and map it into
131 * kernel virtual memory space. Create an entry under /proc file system
132 * and register file operations for that entry.
133 */
134 static int myled_probe(struct platform_device *pdev)
135 {
136     struct proc_dir_entry *myled_proc_entry;
137     int ret = 0;
138
139     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
140     if (!res) {
141         dev_err(&pdev->dev, "No memory resource\n");
142         return -ENODEV;
143     }
144
145     remap_size = res->end - res->start + 1;
146     if (!request_mem_region(res->start, remap_size, pdev->name)) {
147         dev_err(&pdev->dev, "Cannot request IO\n");
148         return -ENXIO;
149     }
150
151     base_addr = ioremap(res->start, remap_size);
152     if (base_addr == NULL) {
153         dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n",
154             (unsigned long)res->start);
155         ret = -ENOMEM;
156         goto err_release_region;
157     }
158

```

Figure 60. myled.c (Cont.)

```

159     myled_proc_entry = proc_create(DRIVER_NAME, 0, NULL,
160                                   &proc_myled_operations);
161     if (myled_proc_entry == NULL) {
162         dev_err(&pdev->dev, "Couldn't create proc entry\n");
163         ret = -ENOMEM;
164         goto err_create_proc_entry;
165     }
166
167     printk(KERN_INFO DRIVER_NAME " probed at VA 0x%08lx\n",
168           (unsigned long) base_addr);
169
170     return 0;
171
172 err_create_proc_entry:
173     iounmap(base_addr);
174 err_release_region:
175     release_mem_region(res->start, remap_size);
176
177     return ret;
178 }
179
180 /* device match table to match with device node in device tree */
181 static const struct of_device_id myled_of_match[] = {
182     {.compatible = "dglnt,myled-1.00.a"},
183     {}
184 };
185
186 MODULE_DEVICE_TABLE(of, myled_of_match);
187
188 /* platform driver structure for myled driver */
189 static struct platform_driver myled_driver = {
190     .driver = {
191         .name = DRIVER_NAME,
192         .owner = THIS_MODULE,
193         .of_match_table = myled_of_match},
194     .probe = myled_probe,
195     .remove = myled_remove,
196     .shutdown = myled_shutdown
197 };
198
199 /* Register myled platform driver */
200 module_platform_driver(myled_driver);
201
202 /* Module Informations */
203 MODULE_AUTHOR("Diligent, Inc.");
204 MODULE_LICENSE("GPL");
205 MODULE_DESCRIPTION(DRIVER_NAME ": MYLED driver (Simple Version)");
206 MODULE_ALIAS(DRIVER_NAME);
207

```

This is the name as in the device tree for this device
 Letter l in dglnt is lower-case L.

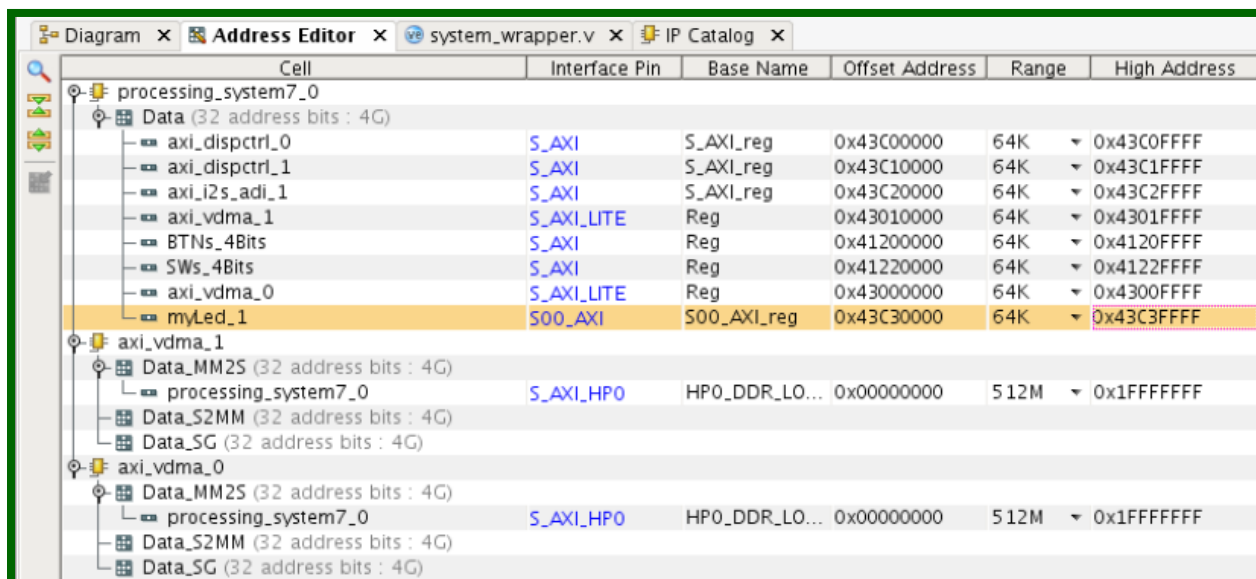
Figure 60. myled.c (Cont.)

4. Compile and generate the driver module using make (as shown in Fig. 61). Don't forget to source Vivado settings.

```
[kfranz@DIGILENT_LINUX drivers]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make -C ../Linux-Digilent-Dev/ M=/home/kfranz/Tutorial/drivers modules
make[1]: Entering directory `/home/kfranz/Tutorial/Linux-Digilent-Dev'
  CC [M]  /home/kfranz//Tutorial/drivers/myLed.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /home/kfranz/Tutorial/drivers/myLed.mod.o
  LD [M]  /home/kfranz/Tutorial/drivers/myLed.ko
make[1]: Leaving directory `/home/kfranz/Tutorial/Linux-Digilent-Dev'
[kfranz@DIGILENT_LINUX drivers]$
```

Figure 61. Compile Driver.

5. We need to add the **myLed** device node into the device tree. Make a copy of the default device tree source in the drivers folder, and modify it according to Fig. 64. The compatibility string of the node is the same as we define in the driver source code (myled.c: line 182). The reg property defines the physical address and size of the node. The address here should match with the address of the myLed IP Core in the address editor tab of the Vivado design, as shown in Fig. 63.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 4G)					
axi_dispctrl_0	S_AXI	S_AXI_reg	0x43C00000	64K	0x43C0FFFF
axi_dispctrl_1	S_AXI	S_AXI_reg	0x43C10000	64K	0x43C1FFFF
axi_i2s_adi_1	S_AXI	S_AXI_reg	0x43C20000	64K	0x43C2FFFF
axi_vdma_1	S_AXI_LITE	Reg	0x43010000	64K	0x4301FFFF
BTNs_4Bits	S_AXI	Reg	0x41200000	64K	0x4120FFFF
SWs_4Bits	S_AXI	Reg	0x41220000	64K	0x4122FFFF
axi_vdma_0	S_AXI_LITE	Reg	0x43000000	64K	0x4300FFFF
myLed_1	S00_AXI	S00_AXI_reg	0x43C30000	64K	0x43C3FFFF
axi_vdma_1					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LO...	0x00000000	512M	0x1FFFFFFF
Data_S2MM (32 address bits : 4G)					
Data_SG (32 address bits : 4G)					
axi_vdma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LO...	0x00000000	512M	0x1FFFFFFF
Data_S2MM (32 address bits : 4G)					
Data_SG (32 address bits : 4G)					

Figure 62. Physical Address for myLed IP Core.

ZYBO should be all lowercase.

```
[kfranz@DIGILENT_LINUX drivers]$ cp ../Linux-Digilent-Dev/arch/arm/boot/dts/zynq-ZYBO.dts ./
[kfranz@DIGILENT_LINUX drivers]$ vim zynq-ZYBO.dts
```

Figure 63. Edit device tree.

```

549         spi-speed-hz = <4000000>;
550         spi-sclk-gpio = <&ps7_gpio_0 59 0>;
551         spi-sdin-gpio = <&ps7_gpio_0 60 0>;
552     };
553
554     myled {
555         compatible = "dglnt,myled-1.00.a";
556         reg = <0x43c30000 0x10000>;
557     };
558 };
559 };

```

Use the offset address you got from
your block design

Figure 64. zynq-ZYBO.dts

6. Recompile the device tree blob as shown in Fig. 65. **Letter l in dglnt is lower-case L.**

```

[kfranz@DIGILENT_LINUX drivers]$ ../Linux-Digilent-Dev/scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb
zynq-ZYBO.dts
DTC: dts->dtb on file "zynq-ZYBO.dts"
[kfranz@DIGILENT_LINUX drivers]$

```

ZYBO should be all lowercase.

Figure 65. Compile DTB.

7. Copy these two files to the first partition of the SD card, as shown in Fig. 66. We are ready to test our driver on-board now.

```

[kfranz@DIGILENT_LINUX drivers]$ ls
devicetree.dtb  Makefile      Module.symvers  myled.ko      myled.mod.o
zynq-ZYBO.dts  modules.order myled.c          myled.mod.c   myled.o
[kfranz@DIGILENT_LINUX drivers]$ cp myled.ko /media/ZYBO_BOOT/d
[kfranz@DIGILENT_LINUX drivers]$ cp devicetree.dtb /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX drivers]$

```

Figure 66. Copy files to SD.

8. Plug the SD card into the ZYBO and we can start testing our driver. Use the **insmod** command to install the driver module into the kernel. After the driver is installed, an entry named **myled** will be created under the **/proc** file system. Writing **0x0F** to **/proc/myled** will light up **LED 0~3**. You can either remove the driver with command **rmmod** or power off the system by command **poweroff**. In both cases, all of the LEDs will be turned off, as shown in Fig. 67. For instructions on using the terminal with the ZYBO, please refer to **Section 5, Step 4** or the Section **Boot from SD** in *Getting Started with Embedded Linux – ZYBO*.

U-Boot 2012.04.01-dirty (June 30 2014 - 12:52:36)

```
DRAM: 512 MiB
WARNING: Caches not enabled
MMC: SDHCI: 0
Using default environment
...
reading uImage
```

```
2457328 bytes read
reading devicetree.dtb
```

```
9728 bytes read
reading uramdisk.image.gz
```

```
3694108 bytes read
```

```
## Starting application at 0x00008000 ...
```

```
Uncompressing Linux... done, booting the kernel.
```

```
[ 0.000000] Booting Linux on physical CPU 0
```

```
[ 0.000000] Linux version 3.6.0-digilent-13.01-00002-g06b3889 (kfranz@DIGILENT_LINUX)
```

```
(gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-79) ) #1 SMP PREEMPT Sun June 30
```

```
23:54:12 PST 2014
```

```
...
```

```
rcS Complete //use fdisk -l to see the microSD card.
```

```
zynq> mount /dev/mmcblk0p1 /mnt/
```

```
zynq> cd /mnt/
```

```
zynq> ls
```

```
BOOT.BIN          devicetree.dtb      uramdisk.image.gz
```

```
myled.ko          uImage
```

```
zynq> insmod myled.ko
```

```
[ 122.160000] myled probed at va 0xe0d20000
```

```
zynq> ls /proc
```

```
1          567          9          fs          partitions
10         582          asound      interrupts   scsi
11         588          buddyinfo   iomem        self
12         594          bus          ioports      slabinfo
13         595          cmdline     irq           softirqs
14         596          config.gz   kallsyms     stat
15         6          consoles    kmsg          swaps
2          608          cpu          kpagecount   sys
3          614          cpuinfo     kpageflags   sysvipc
317        615          crypto      loadavg       timer_list
318        621          device-tree locks         tty
333        641          devices     meminfo       uptime
4          642          diskstats   misc          version
429        643          dma          modules       vmallocinfo
440        647          dri          mounts        vmstat
441        652          driver      mtd           zoneinfo
5          653          execdomains myled
515        7          fb          net
548        8          filesystems pagetypeinfo
```

```
zynq> echo 0x0F > /proc/myled
```

```
zynq> cat /proc/myled
```

```
0x0f
```

```
zynq> mkdir -p /lib/modules/`uname -r`
```

```
zynq> cp myled.ko /lib/modules/`uname -r`
```

```
zynq> rmmod myled
```

```
zynq> ls
BOOT.bin          myled.ko
System Volume Information uImage
devicetree.dtb    uramdisk.image.gz
zynq> insmod myled.ko
insmod: can't insert 'myled.ko': File exists
zynq> _
```

Two errors caused this: word dgInt, the middle letter is lower case L. base address has 4 zeros: 0x43c30000

```
zynq> fdisk -l
Disk /dev/mmcblk0: 8068 MB, 8068792320 bytes
4 heads, 16 sectors/track, 246240 cylinders
Units = cylinders of 64 * 512 = 32768 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/mmcblk0p1        33          32032     1024000    c   Win95 FAT32 (LBA)
Partition 1 does not end on cylinder boundary
/dev/mmcblk0p2       32033       246240     6854656    c   Win95 FAT32 (LBA)
Partition 2 does not end on cylinder boundary
zynq> _
```

Figure 67. RAMDISK

7 User Application

7.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).

7.2 Instructions

1. In this section, we will write a user application that makes the LEDs blink by writing to /proc/myled. Create a directory named **user_app** in the Tutorial folder, as shown in Fig. 68. Inside the **user_app** directory, we will compose the led_blink.c, as shown in Fig. 69.

```
[kfranz@DIGILENT_LINUX Tutorial]$ mkdir user_app
[kfranz@DIGILENT_LINUX Tutorial]$ ls
devicetree.dtb  drivers  linux-digilent  u-boot-digilent  user_app  ZYBO_base_system
[kfranz@DIGILENT_LINUX Tutorial]$
```

Figure 68. User_app

```
[kfranz@DIGILENT_LINUX user_app]$ vim led_blink.c
```

Figure 69. led_blink

```
0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     FILE* fp;
7     while(1) {
8         fp = fopen("/proc/myled", "w");
9         if(fp == NULL) {
10             printf("Cannot open /proc/myled for write\n");
11             return -1;
12         }
13         fputs("0x0F\n", fp);
14         fclose(fp);
15         sleep(1);
16         fp = fopen("/proc/myled", "w");
17         if(fp == NULL) {
18             printf("Cannot open /proc/myled for write\n");
19             return -1;
20         }
21         fputs("0x00\n", fp);
22         fclose(fp);
23         sleep(1);
24     }
25     return 0;
26 }
```

Figure 70. led_blink.c

2. Compose a Makefile and compile led_blink.c into led_blink.o, as shown in Figs. 71-73.

```
[kfranz@DIGILENT_LINUX user_app]$ vim Makefile
```

Figure 71. Makefile.

```
1 CC = arm-xilinx-linux-gnueabi-gcc
2 CFLAGS = -g
3
4 all : led_blink
5
6 led_blink : led_blink.o
7     ${CC} ${CFLAGS} $^ -o $@
8
9 clean :
10     rm -rfv *.o
11     rm -rfv led_blink
12
13 .PHONY : clean
```

Remember tabs not spaces for the indents, really this will cause it to not compile

Figure 72. Makefile.

```
[kfranz@DIGILENT_LINUX user_app]$ make
arm-xilinx-linux-gnueabi-gcc -g -c -o led_blink.o led_blink.c
arm-xilinx-linux-gnueabi-gcc -g -o led_blink led_blink.o
[kfranz@DIGILENT_LINUX user_app]$ ls
led_blink led_blink.c led_blink.o Makefile
[kfranz@DIGILENT_LINUX user_app]$
```

Figure 73. Compile led_blink.

3. Insert the SD card into the computer, and copy the binary file led_blink onto the first partition of SD card, as shown in Fig. 74.

```
[kfranz@DIGILENT_LINUX user_app]$ cp led_blink /media/ZYBO_BOOT/
```

Figure 74. Move led_blink.

```
...
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
zynq> cd /mnt/
zynq> ls
BOOT.BIN          devicetree.dtb      led_blink
myled.ko           ramdisk8M.image.gz  zImage
zynq> insmod myled.ko
[ 122.160000] myled probed at va 0x8000
zynq> ./led_blink
^C
zynq> mkdir -p /lib/modules/`uname -r`
zynq> cp myled.ko /lib/modules/`uname -r`
zynq> rmmod myled
```

Figure 75. RAMDISK.

Here is a linking error when led_blink.o is linked to system library.

```
song@song-VirtualBox:~/user_app$ arm-xilinx-linux-gnueabi-gcc -g -o led_blink led_blink.o
/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../lib/gcc/arm-xilinx-linux-gnueabi/4.8.3
/../../../../arm-xilinx-linux-gnueabi/bin/ld:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../../../../arm-xilinx-linux-gnueabi/libc/lib/libc.so.6: file format not recognized; treating as linker script
/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../lib/gcc/arm-xilinx-linux-gnueabi/4.8.3
/../../../../arm-xilinx-linux-gnueabi/bin/ld:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../../../../arm-xilinx-linux-gnueabi/libc/lib/libc.so.6:1: syntax error
collect2: error: ld returned 1 exit status
song@song-VirtualBox:~/user_app$ arm-xilinx-linux-gnueabi-gcc -g -o led_blink led_blink.o
```

Solution: (1) Go to directory where libc.so.6 resides. (2) Find out what is in the file. It should be the link that this file was supposed to point to. It is libc-2.18.so. (3) Delete libc.so.6 file. (4) run `ln -s libc-2.18.so libc.so.6`

```
Try: sudo apt install <selected package>
song@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$ gedit libc.so.6
```



```
song@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$ ln -s libc-2.18.so libc.so.6
```

Solution: (1) Go to directory where ld-linux.so.3 resides. (2) Find out what is in the file. It should be the link that this file was supposed to point to. It is ld-2.18.so. (3) Delete ld-linux.so.3 file. (4) run `ln -s ld-2.18.so ld-linux.so.3`

```
song@song-VirtualBox:~/user_app$ make
arm-xilinx-linux-gnueabi-gcc -g led_blink.o -o led_blink
/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../lib/gcc/arm-xilinx-linux-gnueabi/4.8.3
/../../../../arm-xilinx-linux-gnueabi/bin/ld:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../../../../arm-xilinx-linux-gnueabi/libc/lib/ld-linux.so.3: file format not recognized; treating as linker script
/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../lib/gcc/arm-xilinx-linux-gnueabi/4.8.3
/../../../../arm-xilinx-linux-gnueabi/bin/ld:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/bin/../../../../arm-xilinx-linux-gnueabi/libc/lib/ld-linux.so.3:1: syntax error
collect2: error: ld returned 1 exit status
Makefile:7: recipe for target 'led_blink' failed
make: *** [led_blink] Error 1
song@song-VirtualBox:~/user_app$ cd /opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib/
song@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$ gedit ld-linux.so.3
```



```
ld-2.18.songsong@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$ rm ld-linux.so.3
song@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$ ln -s ld-2.18.so ld-linux.so.3
song@song-VirtualBox:/opt/Xilinx/SDK/2014.4/gnu/arm/lin/arm-xilinx-linux-gnueabi/libc/lib$
```