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.](image-url)
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.

![Hardware System Architecture](image.png)

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 Digilent 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.

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/BOOT.bin
   creating: zybo_base_system/source/
   creating: zybo_base_system/source/ioe/
   creating: zybo_base_system/source/ioe/hw/
   creating: zybo_base_system/source/ioe/hw/data/
   inflating: zybo_base_system/source/ioe/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.

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

***** Vivado v2014.1 (64-bit)
   **** SW Build 081834 on Fri Apr 4 14:00:25 MDT 2014
   **** IP Build 877525 on Fri Mar 28 15:29:15 MDT 2014
   ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

startGui
```

*Figure 4. Open the Project.*
3. 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.
4. (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.](image-url)
5. 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).
6. The next window will be the Add Interfaces Window. This will create the AX14 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.
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.

![Create And Package New IP](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.
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).
10. Next, we need to connect the user logic to myLed. In the project manager select the file myLed_v0. 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).

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

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

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

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).
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 (Optional)

2.1  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.

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.

   ![Figure 31. U-Boot repository.](image1)

   ![Figure 32. Next-repository.](image2)
2. 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.

```bash
[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-
  zynq_zybo_config
Configuring for zynq ZYBO board...
```

3. 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.

```bash
[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]$ ls
```
3 Generate BOOT.BIN

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.](image)

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.
3. 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 37. Set SDK Workspace Path.**

**Figure 38. Export hardware design to SDK.**
4. In the New Project window, select Xilinx->Application Project, and then Click Next (Fig. 40).

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. Click Next, as shown in Fig. 41.
6. Select Zynq FSBL as template, and click Finish as shown in Fig. 42.

Figure 41. New Application Project.

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).

![Figure 43. fsbl_hooks.c](image)

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:

   `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.
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: `/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 44. Create Zynq Boot Image.](image)

![Figure 45. Create Zynq Boot Image Configuration.](image)
11. The created BIN file was named BOOT.bin.

4  Compile Linux Kernel

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.

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:

   **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.

   ```bash
   [kfranz@DIGILENT_LINUX ~]$ git clone https://github.com/DigilentInc/Linux-Digilent-Dev.git
   Initialized empty Git repository in /home/kfranz/Linux-Digilent-Dev/.git/
   remote: Counting objects: 3586185, done.
   remote: Compressing objects: 100% (3549192/3549192), done.
   remote: Total 3586185 (delta 3067223), reused 3586185 (delta 3067223)
   Receiving objects: 100% (3586185/3586185), 864.81 MiB | 2.76 MiB/s, done.
   Resolving deltas: 100% (3007223/3007223), done.
   [kfranz@DIGILENT_LINUX ~]$  
   ```
   
   ![Figure 46. Clone Kernel.](image)

2. We will start to configure the kernel with the default configuration for ZYBO. The configuration is located at arch/arm/configs/xylinx_zynq_defconfig. To use the default configuration, you can follow Fig. 47.

   ```bash
   [kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabihf- xilinx_zynq_defconfig
   # configuration written to .config
   #
   [kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$  
   ```
   
   ![Figure 47. Default Configuration.](image)

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

   ```bash
   [kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabihf-
   scripts/kconfig/conf --silentoldconfig Kconfig
   CHK include/config/kernel.release
   CHK include/generated/uapi/linux/version.h
   CHK include/generated/utsrelease.h
   ```
   
   ![Result of compiling Linux Kernel](image)
4. After the compilation, the kernel image is located at arch/arm/boot/zImage. However, in this case the kernel image has to be a uImage (unzipped) rather than a zimage. To make the uimage, follow Fig. 49.

```bash
$ kfranz@DIGILENT_LINUX Linux-Digilent-Dev]
$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabihf- UIMAGE_LOADADDR=8x8000 uImage
```

**Figure 49. Create uImage.**

**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.

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

**Figure 50. Change Path.**

## 5 Test Kernel Image with Pre-built File System

### 5.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page](#).
- Pre-built File System Image: ramdisk Image is available in ZYBO Linux Reference Design.
- BOOT.BIN from Section 3, uImage from Section 4.

### 5.2 Instructions

1. To boot the Linux operating system on the ZYBO, you need BOOT.BIN, a Linux kernel image (uImage), a device tree blob (DTB file), and a file system. BOOT.BIN has been created in Section 3 and uimage 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.

```c
48 chosen {
49     /* bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
50         rootfstype=ext4 rootwait devtmpfs.mount=1"; */
51     bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M
52         init=/init earlyprintk rootwait devtmpfs.mount=1";
53         linux,stdout-path="/axi@0/serial@e0001000";
54 }
```

**Figure 51. Edit device tree.**

2. Generate DTB file, as shown in Fig. 53.

```bash
[kfranz@DIGILENT LINUX Tutorial]$ ./scripts/dtc/dtc -I dts -O dtb -o ./devicetree.dtb arch/arm/boot/dts/zynq-ZYBO.dts
[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
[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]$ cp
```

**Figure 53. Generate DTB File.**

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.

```bash
[kfranz@DIGILENT LINUX Tutorial]$ ls
devicetree.dtb Linux-Digilent-Dev
[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
[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/
```

**Figure 54. Ramdisk.**

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.
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 Digilent GitHub repository https://github.com/Digilentinc/Linux-Digilent-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.

   ![Figure 56. Driver Directory.](image)

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

   ![Figure 57. Create Makefile.](image)

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

```makefile
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.

**Note:** make sure the spacing in the Makefile is made up of tabs, not spaces, where necessary. Then hit esc to exit insert mode and `:x` to save the file and exit vim editor.
```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/uaccess.h>  /* Needed for copy_from_user */
#include <asm/io.h>       /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h> /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h> /* Needed for Platform Driver Functions */

/* Define Driver Name */
#define DRIVER_NAME "myled"

unsigned long *base_addr; /* Virtual Base Address */
struct resource *res;    /* Device Resource Structure */
unsigned long remap_size; /* Device Memory Size */

/ Write operation for /proc/myled
* ___________________________
* When user cat a string to /proc/myled file, the string will be stored in
* char __user *buf. This function will copy the string from user
* space into kernel space, and change it to an unsigned long value.
* It will then write the value to the register of myled controller,
* and turn on the corresponding LEDs eventually.
*/
static ssize_t proc_myled_write(struct file *file, const char __user * buf,
                 size_t count, loff_t * ppos)
{
    char myled_phrase[16];
    u32 myled_value;

    if (count < 11) {
        if (copy_from_user(myled_phrase, buf, count))
            return -EFAULT;

        myled_phrase[count] = '\0';
    }

    myled_value = simple_strtoul(myled_phrase, NULL, 0);
    wmb();
    iowrite32(myled_value, base_addr);
    return count;
}

/* Callback function when opening file /proc/myled
* ___________________________
* Read the register value of myled controller, print the value to
* the sequence file struct seq_file *p. In file open operation for /proc/myled
* this callback function will be called first to fill up the seq_file,
* and seq_read function will print whatever in seq_file to the terminal.
*/
static int proc_myled_show(struct seq_file *p, void *v)
{
    u32 myled_value;
    myled_value = ioread32(base_addr);
    seq_printf(p, "0x%x", myled_value);
    return 0;
}
```

Figure 60. myled.c
/* Open function for /proc/myled
*  ----------------------------
*  When user want to read /proc/myled (i.e. cat /proc/myled), the open function
*  will be called first. In the open function, a seq_file will be prepared and the
*  status of myled will be filled into the seq_file by proc_myled_show function.
*  */
static int proc_myled_open(struct inode *inode, struct file *file)
{
    unsigned int size = 16;
    char *buf;
    struct seq_file *m;
    int res;
    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (!buf)
        return -ENOMEM;
    res = single_open(file, proc_myled_show, NULL);
    if (!res) {
        m = file->private_data;
        m->buf = buf;
        m->size = size;
    } else {
        kfree(buf);
    }
    return res;
}

/* File Operations for /proc/myled */
static const struct file_operations proc_myled_operations = {
    .open = proc_myled_open,
    .read = seq_read,
    .write = proc_myled_write,
    .llseek = seq_lseek,
    .release = single_release
};

/* Shutdown function for myled
*  ----------------------------
*  Before myled shutdown, turn-off all the leds
*/
static void myled_shutdown(struct platform_device *pdev)
{
    iowrite32(0, base_addr);
}

Figure 60. myled.c (Cont.)
/* Remove function for myled */
* ----------------------------------
* When myled module is removed, turn off all the leds first,
* release virtual address and the memory region requested.
*/

static int myled_remove(struct platform_device *pdev)
{
    myled_shutdown(pdev);
    /* Remove /proc/myled entry */
    remove_proc_entry(DRIVER_NAME, NULL);
    /* Release mapped virtual address */
    iounmap(base_addr);
    /* Release the region */
    release_mem_region(res->start, remap_size);
    return 0;
}

/* Device Probe function for myled */
* ------------------------------------
* Get the resource structure from the information in device tree.
* request the memory region needed for the controller, and map it into
* kernel virtual memory space. Create an entry under /proc file system
* and register file operations for that entry.
*/

static int myled_probe(struct platform_device *pdev)
{
    struct proc_dir_entry *myled_proc_entry;
    int ret = 0;
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res) {
        dev_err(&pdev->dev, "No memory resource\n");
        return -ENODEV;
    }
    remap_size = res->end - res->start + 1;
    if (!request_mem_region(res->start, remap_size, pdev->name)) {
        dev_err(pdev->dev, "Cannot request IO\n");
        return -ENXIO;
    }
    base_addr = ioremap(res->start, remap_size);
    if (base_addr == NULL) {
        dev_err(pdev->dev, "Couldn't ioremap memory at 0x%08lx\n",
                (unsigned long)res->start);
        ret = -ENOMEM;
        goto err_release_region;
    }
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  return 0;
170
171  err_create_proc_entry:
172  iounmap(base_addr);
173  err_release_region:
174  release_mem_region(res->start, remap_size);
175  return ret;
176 }
177
178 /* device match table to match with device node in device tree */
179 static const struct of_device_id myled_of_match[] = {
180     {.compatible = "dglnt,myled-1.00.a"},
181     {}},
182 }
183
184 MODULE_DEVICE_TABLE(of, myled_of_match);
185
186 /* platform driver structure for myled driver */
187 static struct platform_driver myled_driver = {
188     .driver = {
189         .name = DRIVER_NAME,
190         .owner = THIS_MODULE,
191         .of_match_table = myled_of_match},
192     .probe = myled_probe,
193     .remove = myled_remove,
194     .shutdown = myled_shutdown
195     },
196
197 /* Register myled platform driver */
198 module_platform_driver(myled_driver);
199
200 /* Module Information */
201 MODULE_AUTHOR("Digilent, Inc.");
202 MODULE_LICENSE("GPL");
203 MODULE_DESCRIPTION(DRIVER_NAME ": MYLED driver (Simple Version)");
204 MODULE_ALIAS(DRIVER_NAME);
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-gnuabi-
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. 62. 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.

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

**Figure 62. Physical Address for myLed IP Core.**

**Figure 63. Edit device tree.**
6. Recompile the device tree blob as shown in Fig. 65.

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

```
[kfranz@DIGILENT_LINUX drivers]$ ls
devicetree.dtbo modules.order myled.ko myled.mod.o
zyq-ZYBO.dts myled.c myled.mod.c myled.o
[kfranz@DIGILENT_LINUX drivers]$ cp myled.ko /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX drivers]$ cp devicetree.dtbo /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX drivers]$
```

```
Figure 64. zynq-ZYBO.dts

Figure 65. Compile DTB.

Figure 66. Copy files to SD.
```

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.

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
zymq> mount /dev/mmcblk0p1 /mnt/
zymq> cd /mnt/
zymq> ls
BOOT.BIN   devicetree.dtb  uramdisk.image.gz
myled.ko   uImage
zymq> insmod myled.ko
[  122.160000] myled probed at va 0xe0d20000
zymq> 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
zymq> echo 0x0F > /proc/myled
zymq> cat /proc/myled
0x0F
zymq> mkdir -p /lib/modules/`uname -r`
zymq> cp myled.ko /lib/modules/`uname -r`
zymq> rmmod myled

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.

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    FILE* fp;
    while(1) {
        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
            printf("Cannot open /proc/myled for write\n");
            return -1;
        }
        fputs("0x0F\n", fp);
        fclose(fp);
        sleep(1);
        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
            printf("Cannot open /proc/myled for write\n");
            return -1;
        }
        fputs("0x00\n", fp);
        fclose(fp);
        sleep(1);
    }
    return 0;
}
```

Figure 68. User_app

Figure 69. led_blink

Figure 70. led_blink.c
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
zyng> mount /dev/mmcblk0p1 /mnt/
zyng> cd /mnt/
zyng> ls
BOOT.BIN  devicetree.dtb  led_blink
myled.ko  ramdisk8M.image.gz  zImage
zyng> insmod myled.ko
[ 122.160000] myled probed at va 0x8000
zyng> ./led_blink
^C
zyng> mkdir -p /lib/modules/`uname -r`
zyng> cp myled.ko /lib/modules/`uname -r`
zyng> rmmod myled
```

Figure 75. RAMDISK.