Tutorial of DE1-SoC Board

0 Preparing for using the DE1-SoC Board

You will be requested to finish the assignments on Linux using the DE1-SoC Board. This tutorial uses vmware to create a virtual image of Linux system. If you already have a virtual image available or you are working on a Linux machine, you can skip this section.

0.1 Download and Install Vmware and Ubuntu

Vmware Fusion is available on the school website.
This is the website: https://www.csc.ncsu.edu/vmap/

After installing Vmware Fusion correctly, download the Ubuntu Desktop.
This is the website: https://www.ubuntu.com/download/desktop

Now you have vmware and a Linux system image available.

0.2 Create an ubuntu virtual image

Launch Vmware Fusion, click Add/new and then you will see the window shown in Figure 1. Select install from disc or image and continue.

Then choose the ubuntu image you downloaded before and continue.

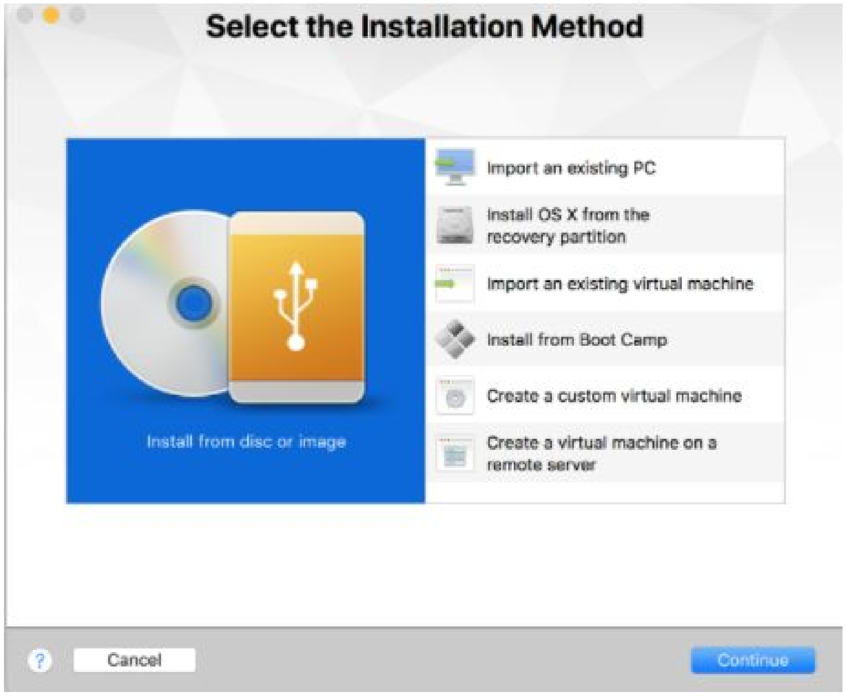

Figure 1. Window shown in Vmware Fusion of installation

Figure 2. Window in Vmware Fusion shown selecting system image

Set up your own account and password and finally the virtual image is created.

0.3 Preparing the Linux MicroSD Card

The DE1-SoC board is designed to boot Linux from an inserted Linux microSD card. In this section, you will prepare a Linux microSD card by placing the *DE1-SoC-UP-Linux.img* image file into a microSD card. This section assumes that you have access to a computer with a microSD card reader. To write the image into the microSD card, you can use the Win32 Disk Imager tool which can be downloaded from the Internet or other tools you prefer.

This is the link of DE1-SoC-UP-Linux.img:
https://drive.google.com/open?id=1mRGLKKx2EI6iIaF7Q4BnS09uAdKXMYyL

The instructions using this tool are provided below:

1. Insert the microSD card into your computer using a microSD card reader, then launch Win32 Disk Imager.
2. Select the drive letter corresponding to the microSD card under Device.
3. Select the *DE1-SoC-UP-Linux.img* image under Image File. This image can be downloaded from the above link.
4. Click Write to write the microSD card. If prompted to confirm the overwrite, press yes.

1 Running Linux on the DE1-SoC Board

1.1 Configuring the DE1-SoC Board for use with Linux

First ensure that the DE1-SoC Board is powered off, then insert the Linux microSD card into the microSD card slot which is right next to the USB port. Before turning on the board, configure the MODE SELECT switches found on the underside of the board to match the settings shown in Figure 3.

1.2 Connecting the DE1-SoC Board to the Host Computer

Before booting the board, you should first connect the DE1-SoC Board to your host computer. The board can communicate with the host computer in two methods: using a USB cable to

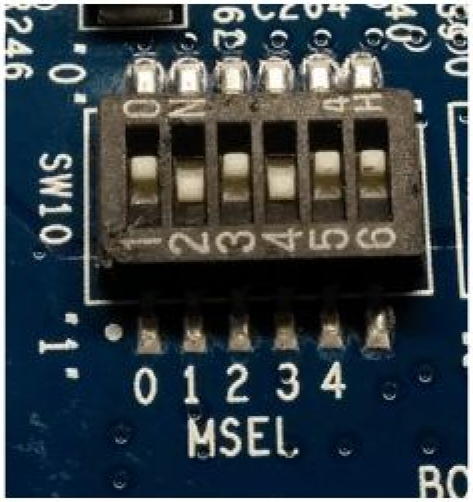

Figure 3. Configuring the MSEL switches of the DE1-SoC board

connect to a Linux command-line prompt, or using a network connection to connect to a Linux graphical user interface (GUI). Each method will be described below.

Figure 4 shows the USB and power cables.

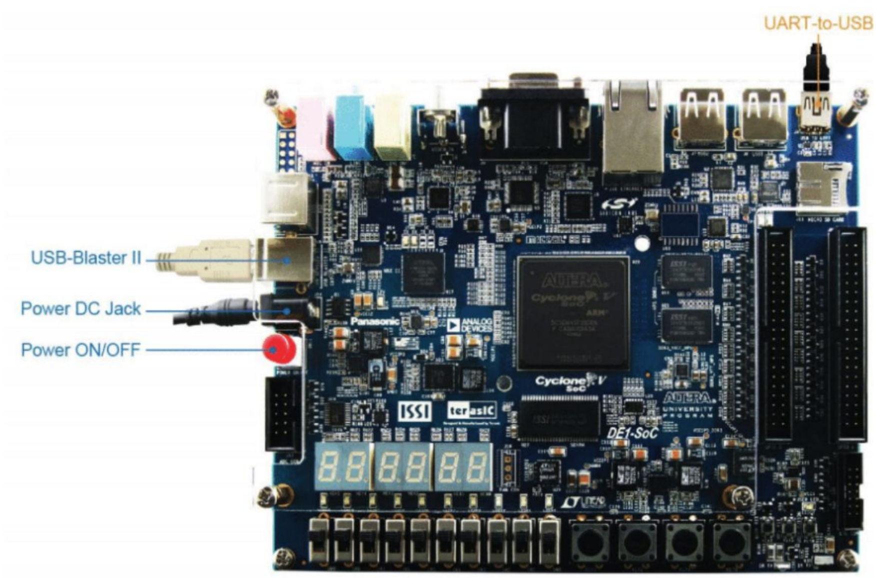

Figure 4. USB and Power Cables

1.3 Connecting to the Host Computer using a USB cable

On the DE1-SoC Board, the HPS's UART is attached to a UART-to-USB chip that can be connected to a host computer using a USB cable. On the host computer, we can use a *terminal* program to display this text. For this tutorial, we will be using **Putty**.

On a Linux computer serial communication devices such as the UART-to-USB are treated as teletype (TTY) devices. Since there could be multiple TTY devices connected to the PC, each TTY device is assigned a unique identifier. The name assigned to your UART-to-USB connection can be determined by running the command `dmesg | grep tty` as shown in Figure 5. In Figure 5, you can see the UART-to-USB chip (FTDI USB Serial Device converter) has been assigned the name ttyUSB0.

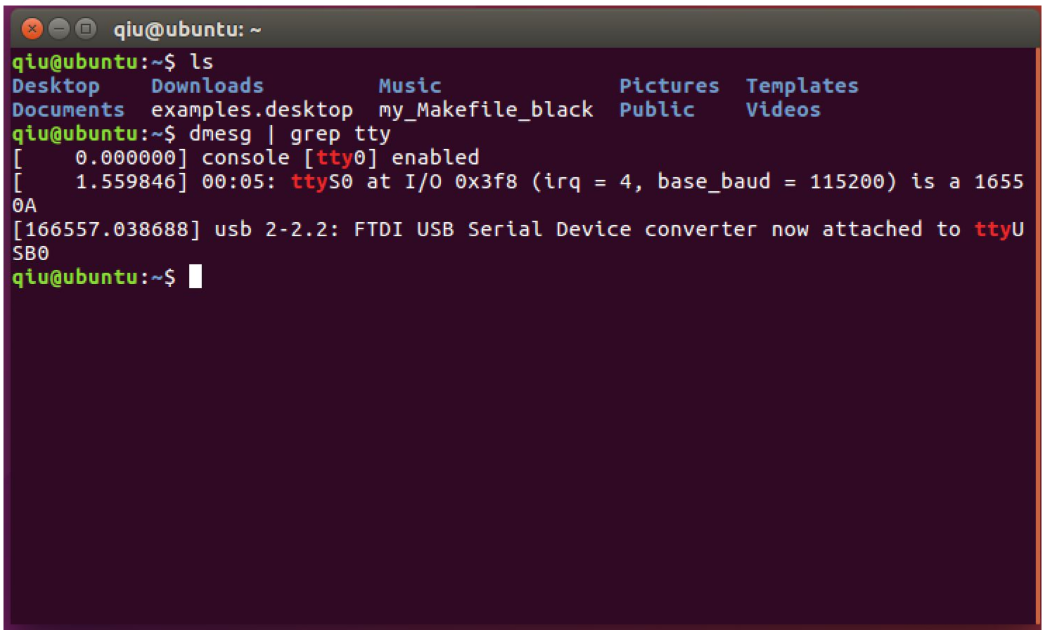

Figure 5. Determining the TTY device that corresponds to the UART-to-USB connection

Type command `putty` in the terminal to invoke putty. Enter the information as in Figure 6 shown.

You might encounter an error shows that: Unable to open connection to: Unable to open serial port. One solution to this is installing gksu by using command `sudo apt-get install gksu`. Then invoke putty by command `gksu putty`.

Turn on the board by pressing the red button, and then press open in the putty window. Once the board finishes booting, you will be logged in to the Linux command line interface as the *root* user. Press *Enter* on your keyboard to see that the interface responds. Type a Linux command such as ls, which shows a listing of directories and files. Figure 7 shows the command line interface after booting.

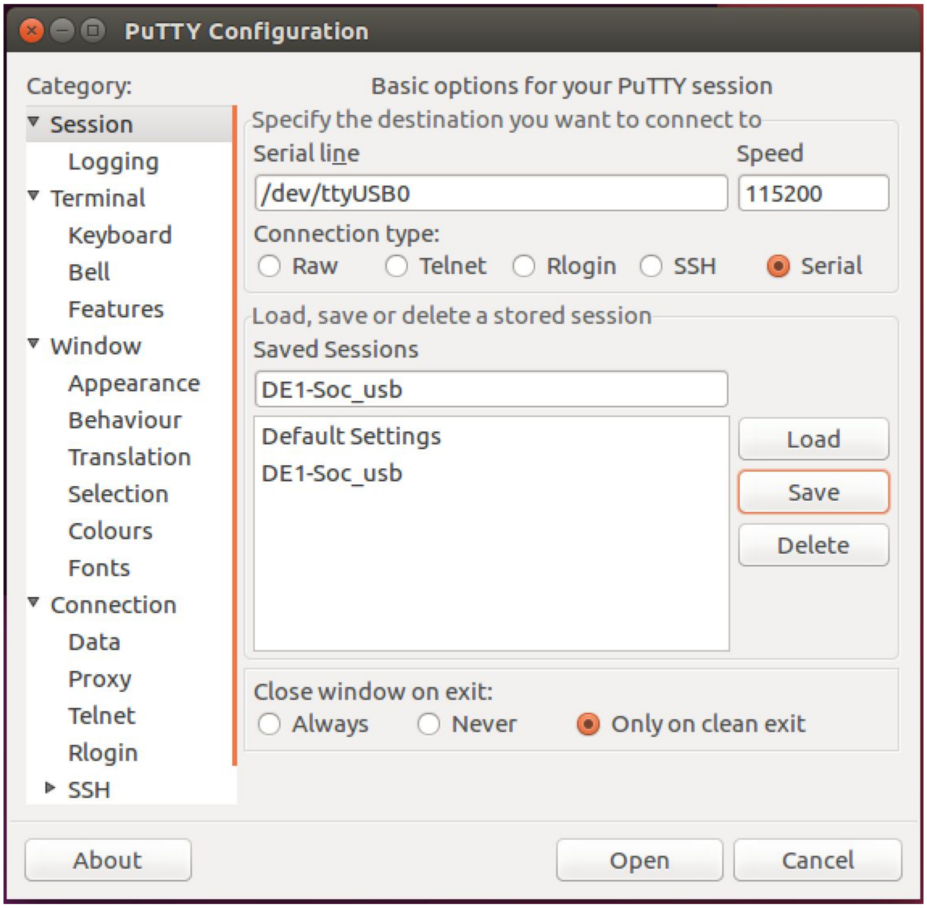

Figure 6. Putty's main window

Figure 7. The Linux command line prompt showing the root ('#') logon

1.4 Connecting to the Host Computer using a WiFi Adapter

To make a network connection to the De1-SoC board using a WiFi adapter, you first have to connect your host computer to the board via a USB cable, as described in section 1.3. Then you can use a Terminal window connect to Linux on the board to connect to the desired WiFi network.

Plug the WiFi adapter into a USB port on the DE1-SoC board. In the Terminal window, type the following command lines:

```
cd misc
./connect_wap <ssid> <password>
```

The information inside the angle brackets are the name and password of the WiFi you plan to connect to. The board should become connected to your WiFi network after a few moments.

If you want to use the WiFi at school, connect to the network “**ncsu**”. In the terminal window on the Linux system on the board, type command ifconfig to get the MAC address and register it on the nomad website. After registering the board, you may need to reboot the board. Create a new file under /home/root/misc (I named it connect_ncsu) and copy the following scripts in it.

```
#!/bin/bash
wlan_interface=wlan0
stop network-manager
printf 'network={\n\tssid="ncsu"\n\tkey_mgmt=NONE\n\tpriority=-999\n\t}' >  /home/root/.temp.conf
wpa_supplicant -B -i$wlan_interface -c/home/root/.temp.conf -Dnl80211
printf 'wpa_supplicant done\n'
dhclient wlan0
printf 'dhclient done\n'
ntpdate -s time.nist.gov
rm /home/root/.temp.conf
```

Using the command ./connect_ncsu to make the board online. Then use ifconfig again to the the IP address of the board. Then you can use ssh root@IP address to copy files to the board. Alternatively, you can copy the files to the MicroSD card.

2. Developing Linux Programs for the DE1-SoC Board

In this section, you will learn how to develop your own programs that can run under Linux on the De1-SoC board. The primary method used in this tutorial is called native compilation which means that you write and compile the code using the command-line of the Linux running on the board.

2.1 Native Compilation on the DE1-SoC Board

When a program is compiled on a system to run on the same architecture as that of the system itself, the process is called native compilation. In this section, you will learn how to natively compile a program through the Linux command-line interface, using its built-in compilation toolchain.

To demonstrate native compilation, we will compile a simple “hello world” program. You can compile this program by the following commands:

```
cd helloworld
gcc helloworld.c -o helloworld
```

The gcc command invokes the GNU C Compiler. In our gcc command, we supply two arguments. The first is the source code file, helloworld.c. The second is -o helloworld which tells the compiler to output an executable file named helloworld. Once the compilation in complete, you can run the program by typing ./helloworld. The program outputs the message “Hello World!” then exits.

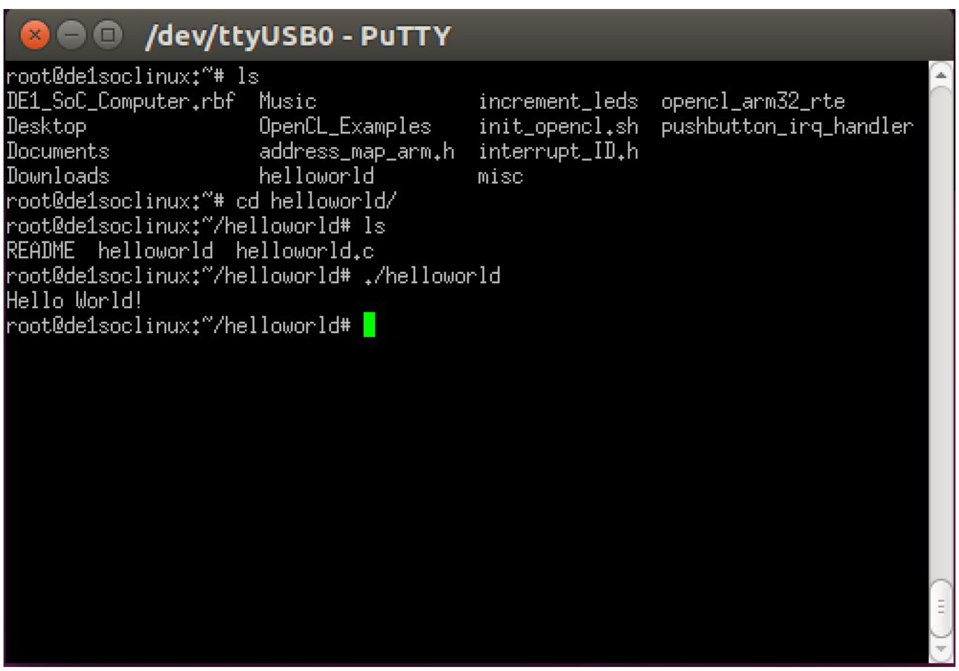

Figure 8. Compiling and executing the helloworld program through the command line

2.2 Accessing Hardware Devices in the FPGA from a Linux Program

Programs running on the ARM processor of the Cyclone V SoC device under Linux can access hardware devices that are implemented in the FPGA through either the *HPS-to-FPGA* or the *Lightweight HPS-to-FPGA* bridge. These bridges are mapped to regions in the ARM memory space. When an FPGA-side component (such as an IP core) is connected to one of these bridges, the component's memory-mapped registers are available for reading and writing by the ARM processor within the bridge's memory region.

If we were developing a baremetal ARM program (a program that does not run on top of an operating system), then accessing peripherals in the FPGA that are mapped to a memory region would be done by simply reading from, or writing to, the appropriate memory address. When programs are being run under Linux it is not as straightforward to access memory-mapped I/O devices. This is because Linux uses a virtual-memory system, and therefore application programs do not have direct access to the processor's physical address space.

To access physical memory addresses from a program running under Linux, you have to call the Linux kernel function mmap and access the system memory device file */dev/mem*. The mmap function, which stands for *memory map*, maps a file into virtual memory. You could, as an example, use mmap to map a text file into memory and access the characters in the text file by reading the virtual memory address span to which the file has been mapped. The system memory device file, */dev/mem*, is a file that represents the physical memory of the computer system. An access into this file at some offset is equivalent to accessing physical memory at the offset address. By using mmap to map the */dev/mem* file into virtual memory, we can map physical addresses to virtual addresses, allowing programs to access physical addresses. In the following section, we will examine a sample Linux program that uses mmap and */dev/mem* to access the Lightweight HPS-to-FPGA (*lwhps2fpga*) bridge's memory span and communicate with an IP core on the FPGA.

2.3 Example Program that uses an FPGA Hardware Device

In this section, we describe an example of code in C that uses a hardware device in the FPGA. The application program uses the lwhps2fpga bridge to alter the state of the red LEDs on the DE1-SoC board. Each time this program is executed, the value displayed on the red LEDs is incremented by one.

DE1-SoC Linux distribution automatically programs the FPGA with the DE1-SoC Computer during boot. The DE1-SoC Computer includes a parallel port that is connected to the red LEDs on the board. This parallel port is connected to the *lwhps2fpga* bridge, which is mapped in the ARM memory space starting at address 0xFF200000. A number of I/O ports are mapped to the bridge's address space, at different *offsets*, and the physical address of any port is given by 0xFF200000 +*offset*. The offset of the red LED port is 0, leading to the address 0xFF200000 + 0x0 = 0xFF200000. The LED parallel port register interface consists of a single register, the *data* register, which can be read to determine the current state of the LEDs, and written to alter the state.

The example code can be found under the directory /home/root/increment_leds. To compile the code, you can type the command: gcc increment_leds.c -o increment_leds. Important lines of the code are described below.

- Lines 2-3 include the fcntl.h and sys/mman.h header files, which are needed to use /dev/mem device file and the mmap and munmap kernel functions.
- Line 4 includes the file address_map_arm.h, which specifies address offsets for all of the FPGA I/O devices that are implemented in the DE1-SoC Computer.
- Lines 7-10 provides prototype declarations for functions that are used to access physical memory. These functions are listed in Figure 10. The functions open_physical and close_physical are used to open/close the /dev/mem device file. The function map_physical calls the mmap kernel function to create a physical-to-virtual address mapping for I/O devices and the unmap_physical closes this mapping. These four functions can be used in any program that needs to access physical memory addresses.
- Line 21 opens the file /dev/mem
- Line 23 maps a part of the /dev/mem file into memory. It maps a portion that starts at the base address of lwhps2fpga(LW_BRIDGE_BASE) and spans LW_BRIDGE_SPAN bytes. The LW_virtual variable will be set to an address that maps to the bottom of the requested physical address space(LW_BRIDGE_BASE). This means an access to LW_virtual + offset will access the physical address 0xFF200000 + offset.
- Line 27 calculates the virtual address that maps to the LED port.
- Line 30 reads the data register of the LED port, increments the value by one and then writes the value back to the register.
- Lines 32-33 unmap and close the /dev/mem file.

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "../address_map_arm.h"

/* Prototypes for functions used to access physical memory addresses */
int open_physical (int);
void * map_physical (int, unsigned int, unsigned int);
void close_physical (int);
int unmap_physical (void *, unsigned int);

/* This program increments the contents of the red LED parallel port */
int main(void)
{
   volatile int * LEDR_ptr;   // virtual address pointer to red LEDs

   int fd = -1;               // used to open /dev/mem
   void *LW_virtual;          // physical addresses for light-weight bridge

   // Create virtual memory access to the FPGA light-weight bridge
   if ((fd = open_physical (fd)) == -1)
      return (-1);
   if ((LW_virtual = map_physical (fd, LW_BRIDGE_BASE, LW_BRIDGE_SPAN)) ==
      NULL)
      return (-1);

   // Set virtual address pointer to I/O port
   LEDR_ptr = (unsigned int *) (LW_virtual + LEDR_BASE);

   // Add 1 to the I/O register
   *LEDR_ptr = *LEDR_ptr + 1;

   unmap_physical (LW_virtual, LW_BRIDGE_SPAN);
   close_physical (fd);
   return 0;
}
```

Figure 9. Source code of increment_LEDs

```c
/* Open /dev/mem to give access to physical addresses */
int open_physical (int fd)
{
   if (fd == -1) // check if already open
      if ((fd = open( "/dev/mem", (O_RDWR | O_SYNC))) == -1)
      {
         printf ("ERROR: could not open \"/dev/mem\"...\n");
         return (-1);
      }
   return fd;
}

/* Close /dev/mem to give access to physical addresses */
void close_physical (int fd)
{
   close (fd);
}

/*
 * Establish a virtual address mapping for the physical addresses starting
 * at base, and extending by span bytes  */
void* map_physical(int fd, unsigned int base, unsigned int span)
{
   void *virtual_base;

   // Get a mapping from physical addresses to virtual addresses
   virtual_base = mmap (NULL, span, (PROT_READ | PROT_WRITE), MAP_SHARED,
       fd, base);
   if (virtual_base == MAP_FAILED)
   {
      printf ("ERROR: mmap() failed...\n");
      close (fd);
      return (NULL);
   }
   return virtual_base;
}

/* Close the previously-opened virtual address mapping */
int unmap_physical(void * virtual_base, unsigned int span)
{
   if (munmap (virtual_base, span) != 0)
   {
      printf ("ERROR: munmap() failed...\n");
      return (-1);
   }
   return 0;
}
```

Figure 10. Functions for managing physical memory addresses

2.4 Device Drives

Device drivers in Linux are software programs that provide an interface to hardware devices. There are two types of device drivers: code that is pre-compiled and distributed with the Linux kernel, and code that is created as a module that can be added to the kernel at runtime. We provide an example of a kernel module in this section.

The kernel module described in this section uses the pushbutton KEYs port in the DE1-SoC Computer.Linux allows interrupts to be used only by software code that is part of the kernel. The ARM processor of the Cyclone V device contains a Generic Interrupt Controller(GIC) which can accomodate 256 interrupt request (IRQ) lines ranging from IRQ0 to IRQ 255. IRQ72 - IRQ 135 are reserved for interrupts originating from hardware devices implemented in side the FPGA. In the DE1-SoC Computer the pushbutton KEYs port is connected to interrupt line IRQ73 which means that our kernel module needs to register an interrupt handler that will respond to IRQ73.

Linux contains drivers for the GIC. allowing us to use high-level interface provided by Linux to register an interrupt handler. The Linux header file linux/interrupt.h provides this interface, among which is the function request_irq(...). This function takes an integer argument irq and a function pointer argument handler, and registers the function as the handler for IRQ number irq.

2.4.1 The Pushbutton Interrupt Handler Kernel Module

The code for our kernel module is shown in Figure 11. Unlike code for normal programs, the kernel module code has no *main* function. Instead, every kernel module has an *init* function which is executed when the module is inserted into the kernel, and an *exit* function which is executed when the module is removed from the kernel.These functions are specified using the macros module_init(...) and module_exit(...).

The *init* function in our module is initialize_pushbutton_handler(void). This function sets the value of the red LED port to *0x200*, which turns on the leftmost LED (as a visual indication that the module has been inserted). It then configures the pushbutton port to start generating interrupts on button presses. Finally, it calls request_irq(...) to register our irq handlerirq_handler(...) to handle pushbutton interrupts. We have included the file interrupt_ID.h which lists all FPGA interrupts in the DE1-SoC Computer.

Once registered, irq_handler(...) is executed whenever the pushbutton port generates an interrupt. The handler does two things. First, it increments the value displayed on the LEDs to provide visual feedback that the interrupt has been handled. Second, it clears the interrupt in the KEYs port by writing to the *edgecapture* register.

In this example, irq_handler(...) serves as a trivial example of an interrupt handler. A “real” driver for a device would do something more useful like transfer data to and from buffers, check the status of devices, and thelike. A device driver that does not use interrupts would still look similar to the code in Figure 11, but without the interrupt-specific code likeirq_handlerandfree_irq.

The *exit* function in our module iscleanup_pushbutton_handler(void). It sets LEDs to 0x0, turning them off, and de-registers the pushbutton irq handler by calling the free_irq(...) function.

2.4.2 Compiling the Kernel Module

The kernel mode source code can be found in the directory /home/root/pushbutton_irq_handler/. To compile the module, use the included Makefile by running the Linux command make.

The contents of the Makefile are shown in Figure 12. The first line specifies the name of the kernel module that is to be built. This line also tells the build system to look for the kernel module code in pushbutton_irq_handler.c and generate the kernel object file pushbutton_irq_handler.ko at the end of the compilation.

The all target, which is the default target when make is run, calls the command make -C ….. The -C argument tells the make program to change work directory to /lib/modules/$(shell uname -r)/build, which is the directory containing the source code and configuration files of the currently running Linux kernel. In this directory is a collection of makefiles called the Linux Kernel Build System (Kbuild) that our make command leverages to build our kernel module. The remaining arguments are used by Kbuild. The argument M=$(PWD) tells Kbuild the location of our kernel module source code and modules tells Kbuild to build a kernel module.

The result of the make command is the generation of the pushbutton_irq_handler.ko kernel module, which is placed in the directory pointed to by the M= argument.

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include "../address_map_arm.h"
#include "../interrupt_ID.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Altera University Program");
MODULE_DESCRIPTION("DE1SoC Pushbutton Interrupt Handler");

void * LW_virtual;                  // Lightweight bridge base address
volatile int *LEDR_ptr, *KEY_ptr;   // virtual addresses

irq_handler_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
   *LEDR_ptr = *LEDR_ptr + 1;
   // Clear the edgecapture register (clears current interrupt)
   *(KEY_ptr + 3) = 0xF;
   return (irq_handler_t) IRQ_HANDLED;
}
static int __init initialize_pushbutton_handler(void)
{
   int value;
   // generate a virtual address for the FPGA lightweight bridge
   LW_virtual = ioremap_nocache (LW_BRIDGE_BASE, LW_BRIDGE_SPAN);

   LEDR_ptr = LW_virtual + LEDR_BASE;  // virtual address for LEDR port
   *LEDR_ptr = 0x200;                  // turn on the leftmost light

   KEY_ptr = LW_virtual + KEY_BASE;    // virtual address for KEY port
   *(KEY_ptr + 3) = 0xF; // Clear the edgecapture register
   *(KEY_ptr + 2) = 0xF; // Enable IRQ generation for the 4 buttons

   // Register the interrupt handler.
   value = request_irq (KEYS_IRQ, (irq_handler_t) irq_handler, IRQF_SHARED,
      "pushbutton_irq_handler", (void *) (irq_handler));
   return value;
}
static void __exit cleanup_pushbutton_handler(void)
{
   *LEDR_ptr = 0; // Turn off LEDs and de-register irq handler
   free_irq (KEYS_IRQ, (void*) irq_handler);
}

module_init(initialize_pushbutton_handler);
module_exit(cleanup_pushbutton_handler);
```

Figure 11. Source code for the pushbutton interrupt handler kernel module

```
obj-m += pushbutton_irq_handler.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Figure 12. Kernel module Makefile

2.4.3 Running the Kernel Module

A kernel module is executed by inserting it into the Linux kernel using the command insmod <filename.ko>. As shown in Figure 13, you can use the command lsmod to confirm that your modules has been loaded. Once the module is inserted, you should see that the leftmost red LED on the DE1-SoC board is turned on. Now press any of the four push buttons to generate an interrupt on IRQ72, and confirm that the value displayed on the LEDs by increments by one.

To stop a kernel module, you can remove it from the kernel by using the command rmmod <module_name>. You can use the lsmod command to confirm that the module has been removed.

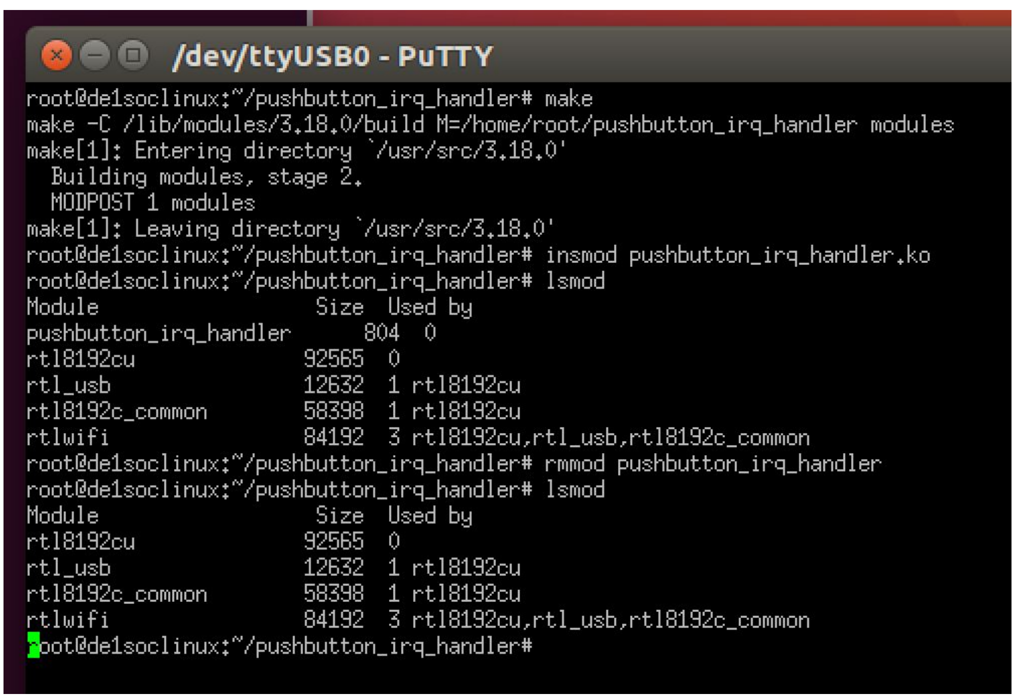

Figure 13. Inserting and removing the kernel module

2.5 Include Files

Figure 14 shows the contents of the include file address_map_arm.h. The file lists memory and FPGA I/O addresses in the DE1-SoC Computer.

```
/* Memory */
#define DDR_BASE              0x00000000
#define DDR_SPAN              0x3FFFFFFF
#define A9_ONCHIP_BASE        0xFFFF0000
#define A9_ONCHIP_SPAN        0x0000FFFF
#define SDRAM_BASE            0xC0000000
#define SDRAM_SPAN            0x03FFFFFF
#define FPGA_ONCHIP_BASE      0xC8000000
#define FPGA_ONCHIP_SPAN      0x0003FFFF
#define FPGA_CHAR_BASE        0xC9000000
#define FPGA_CHAR_SPAN        0x00001FFF

/* Cyclone V FPGA devices */
#define LW_BRIDGE_BASE        0xFF200000

#define LEDR_BASE             0x00000000
#define HEX3_HEX0_BASE        0x00000020
#define HEX5_HEX4_BASE        0x00000030
#define SW_BASE               0x00000040
#define KEY_BASE              0x00000050
#define JP1_BASE              0x00000060
#define JP2_BASE              0x00000070
#define PS2_BASE              0x00000100
#define PS2_DUAL_BASE         0x00000108
#define JTAG_UART_BASE        0x00001000
#define JTAG_UART_2_BASE      0x00001008
#define IrDA_BASE             0x00001020
#define TIMER_BASE            0x00002000
#define AV_CONFIG_BASE        0x00003000
#define PIXEL_BUF_CTRL_BASE   0x00003020
#define CHAR_BUF_CTRL_BASE    0x00003030
#define AUDIO_BASE            0x00003040
#define VIDEO_IN_BASE         0x00003060
#define ADC_BASE              0x00004000

#define LW_BRIDGE_SPAN        0x00005000
```

Figure 14. The contents of address_map_arm.h

Figure 15 shows the contents of the interrupt_ID.h. This file lists the FPGA interrupt line number in the DE1-SoC Computer.

```
/* FPGA interrupts */
#define    INTERVAL_TIMER_IRQ      72
#define    KEYS_IRQ                73
#define    FPGA_IRQ2               74
#define    FPGA_IRQ3               75
#define    FPGA_IRQ4               76
#define    FPGA_IRQ5               77
#define    AUDIO_IRQ               78
#define    PS2_IRQ                 79
#define    JTAG_IRQ                80
#define    IrDA_IRQ                81
#define    FPGA_IRQ10              82
#define    JP1_IRQ                 83
#define    JP2_IRQ                 84
#define    FPGA_IRQ13              85
#define    FPGA_IRQ14              86
#define    FPGA_IRQ15              87
#define    FPGA_IRQ16              88
#define    PS2_DUAL_IRQ            89
#define    FPGA_IRQ18              90
#define    FPGA_IRQ19              91
```

Figure 15. The contents of interrupt_ID.h

3. OpenCL on DE1-SoC Board

This section gives introduction on how to setup OpenCL development environment, compile and execute example projects for DE1-SoC. Users can refer to Altera SDK for OpenCL Programming Guide for more details about OpenCl coding instruction.
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

3.1 Demo of OpenCL on DE1-SoC Board

After you launch into the Linux on DE1-SoC board, type command source ./initopencl.sh to load the OpenCl Linux kernel driver and setup environment variables for OpenCL Run-Time Environment that is already installed on the microSD card.

- Launch hello world demo:

- Type cd to return to the directory /home/root
- Type cd OpenCL_Examples
- Type aocl program /dev/acl0 hello_world.aocx to configure the FPGA with the hello_world kernel
- Type ./hello_world to launch the hello_world host application, as shown in Figure 16.

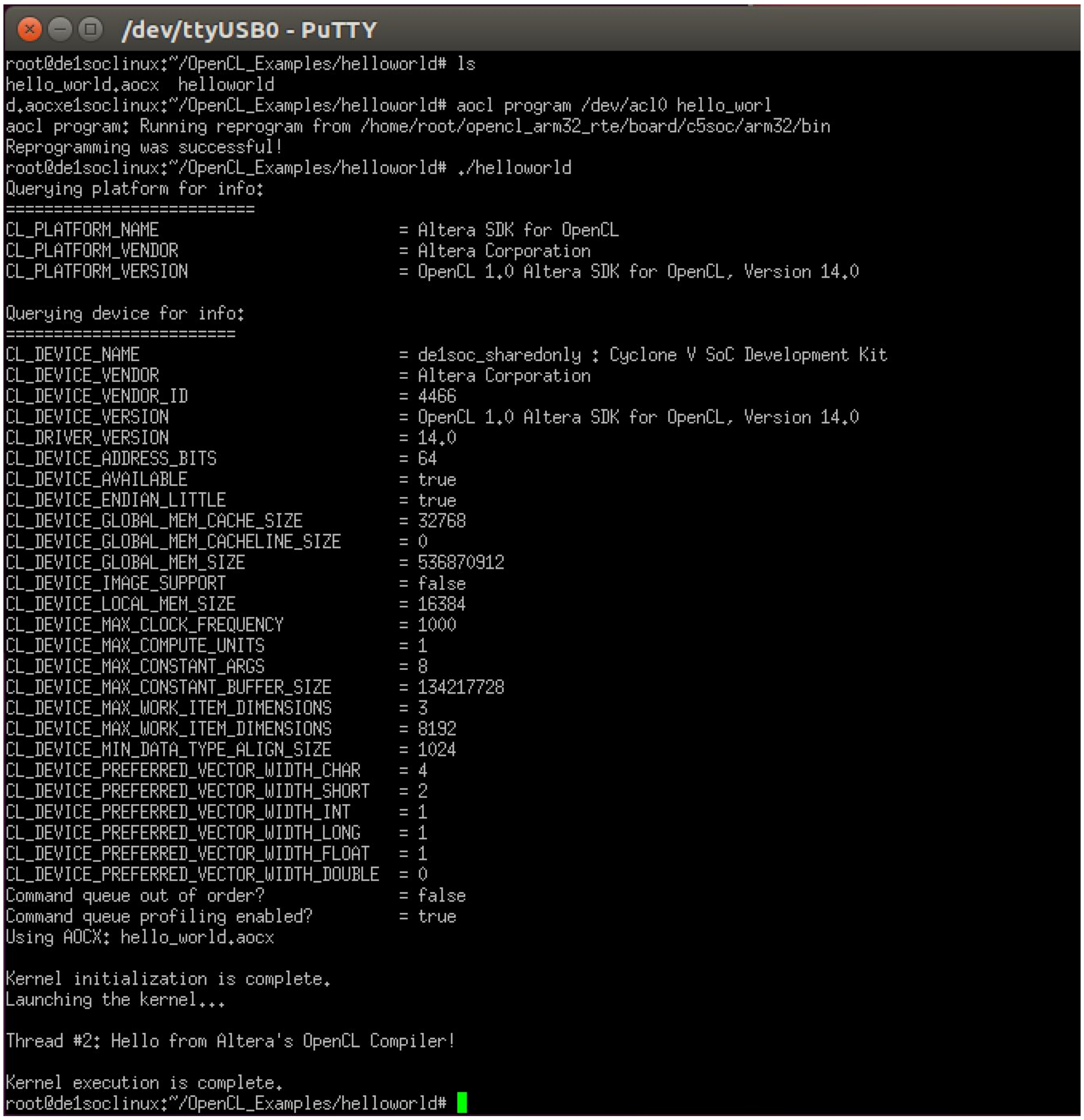

Figure 16. Hello world demo

3.2 Software Installation

3.2.1 Install Altera Quartus II and OpenCL SDK

Altera Quartus II and OpenCL SDK are available from the website:

http://dl.altera.com/opencl

For Quartus II installation, please make sure that the Cyclone V device package is selected. Please choose the right version and download method for your operating system.

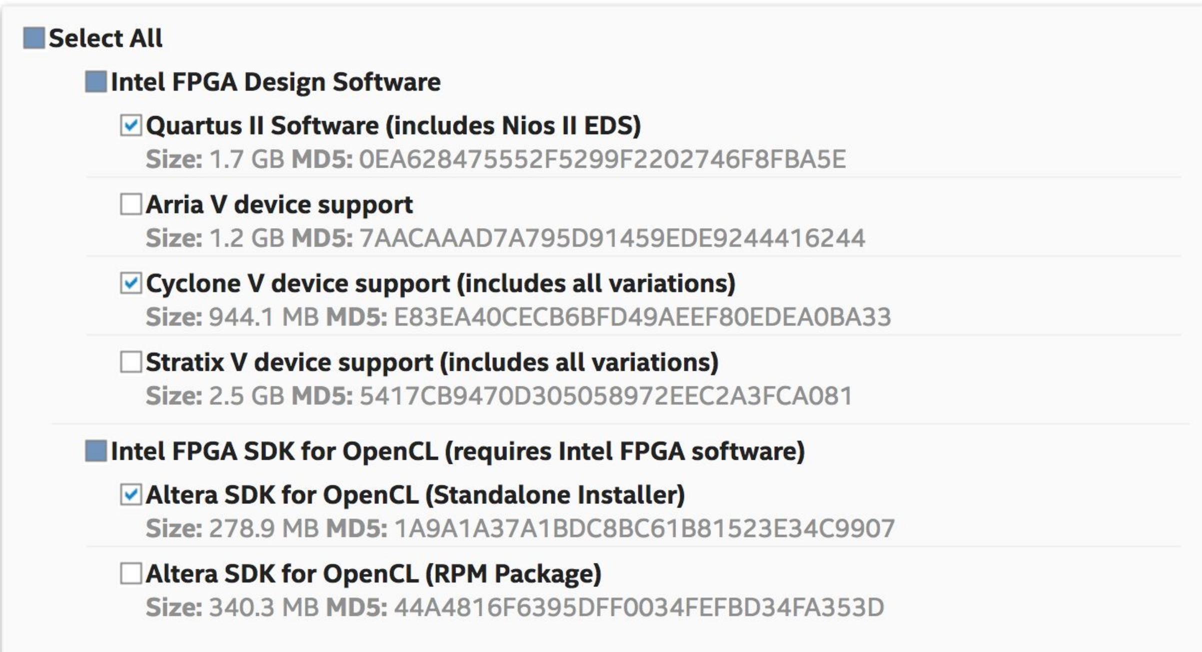

Figure 17. Packages to download to install Quartus II and OpenCL SDK

After the download completes, open a terminal window and change the directory to where you download the packages. Type commands

```
chmod +x *.run
./QuartusSetupWeb-14.0.0.200-linux.run
./AOCLSetup-14.0.0.200-linux.run
```

Follow the instructions from the prompted window. Quartus and OpenCL SDK will be installed.

3.2.2 Install Altera SoC EDS (Optional)

Altera SoC EDS tool is required to cross-compile the host program for ARM processor. The software is available from the website:

http://dl.altera.com/soceds

Please make sure DS-5 is installed during the installation of SoC EDS.

3.2.3 Install DE1-SoC OpenCL Board Support Package (BSP)

After Quartus II and OpenCL SDK are installed, please download the DE1-SoC BSP file **DE1-SoC_openC_BSP.zip from**

http://cd-de1-soc.terasic.com

Unzip the downloaded file and copy the terasic folder to the Altera OpenCL SDK folder.

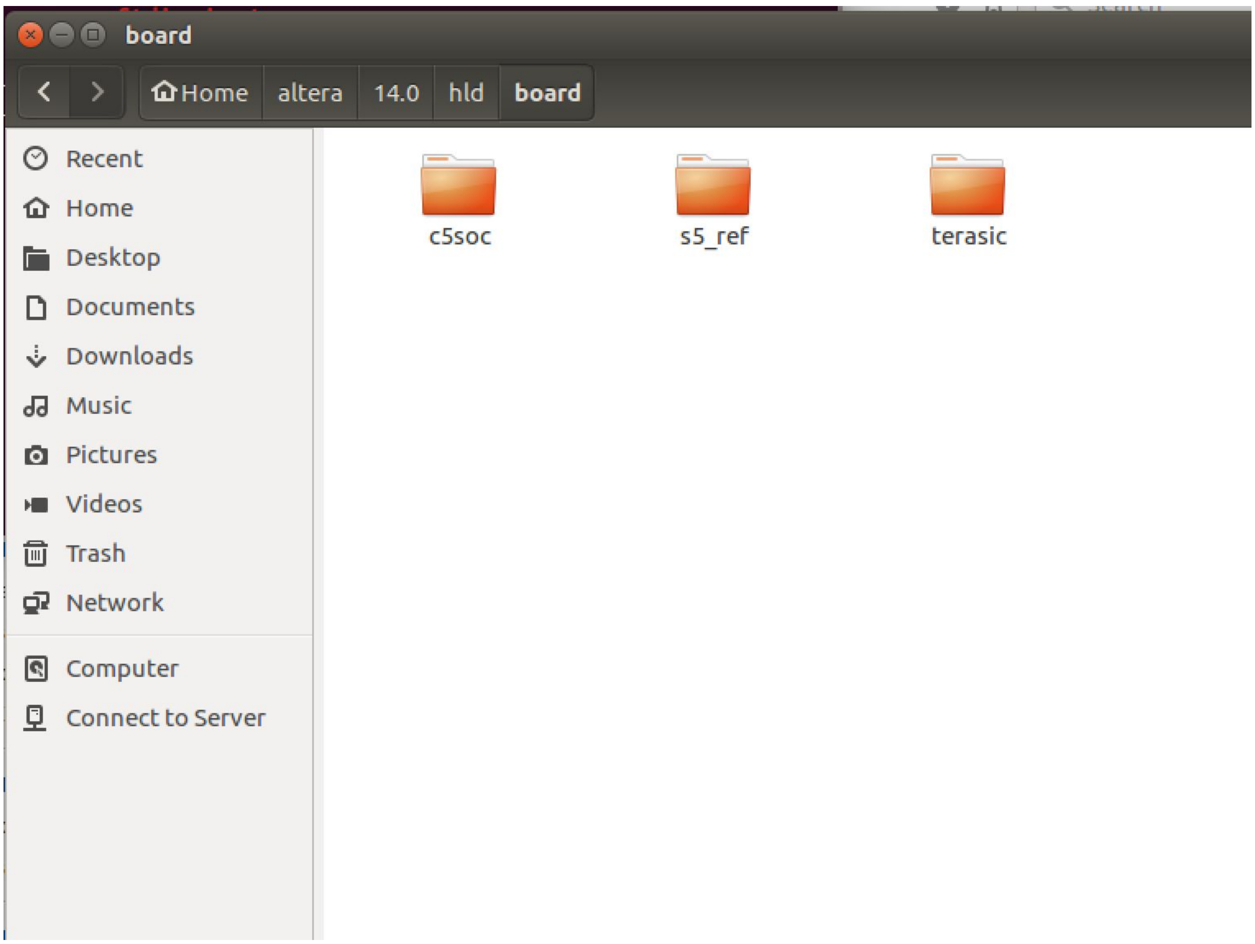

Figure 18. Copy the terasic folder to the board folder

3.2.4 OpenCL License Installation

A license for OpenCL is required to compile OpenCL project with Altera OpenCL SDK. After users have obtained a license file named "license.dat", it needs to be saved to the local disk such as /home/qiu/altera/14.0/hld/license.dat.

3.3 Configuration of Environment Variables

In your root directory, in the terminal window, type command gedit opencl_env and append the following lines into the file.

export QUARTUS_ROOTDIR=/home/qiu/altera/14.0/quartus
export ALTERAOCLSDKROOT=/home/qiu/altera/14.0/hld
export PATH=$PATH:"$QUARTUS_ROOTDIR"/bin:"$ALTERAOCLSDKROOT"/linux64/bin:
"$ALTERAOCLSDKROOT"/bin:/home/qiu/altera/14.0/embedded/ds-5/bin
export LD_LIBRARY_PATH="$ALTERAOCLSDKROOT"/linux64/lib
export AOCL_BOARD_PACKAGE_ROOT="$ALTERAOCLSDKROOT"/board/terasic/de1soc
export QUARTUS_64BIT=1
export LM_LICENSE_FILE=/home/qiu/altera/14.0/hld/license.dat

You need to change the directory to the location where you installed Quartus and OpenCL SDK. After you finish the edit, type source opencl_env in the terminal to apply the settings.

3.4 Verification of OpenCL Environment

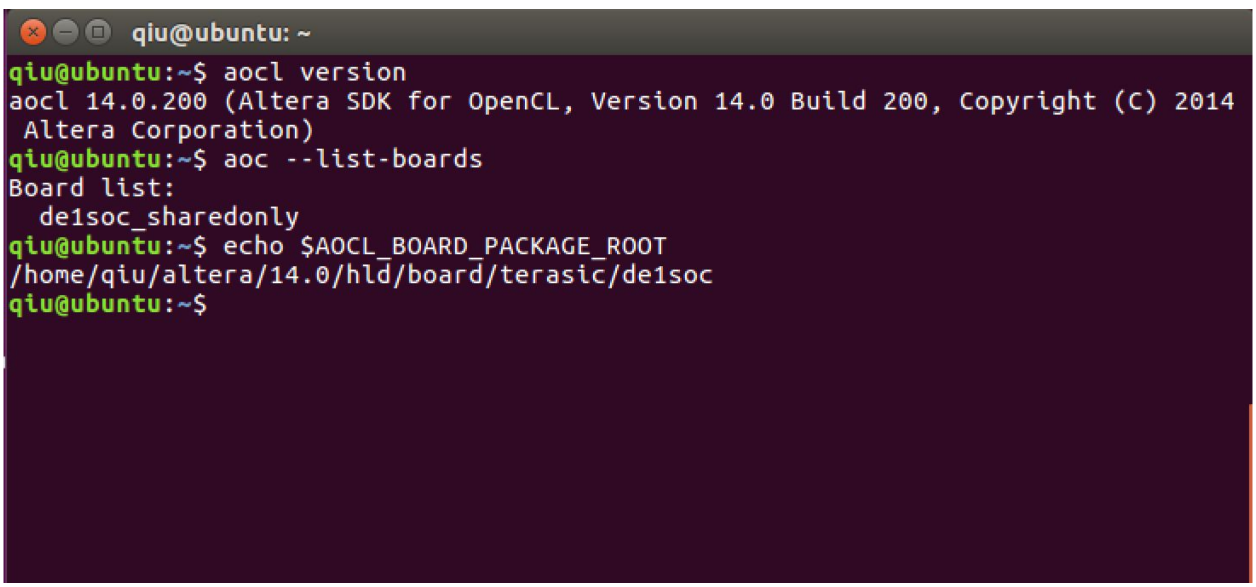

Figure 19. Try the commands in the figure to check the environment variables settings

3.5 Build and Execute OpenCL Project.