

1 Introduction

This tutorial describes a Linux operating system that runs on DE Series boards. Linux runs on the ARM Cortex-A9 processor that is part of the Cyclone V SoC device. In this tutorial we show how Linux can be stored onto a microSD memory card and booted by the ARM processor. We also show how software programs can be developed that run on the ARM processor under Linux, and which can make use of the hardware resources in the Computer System. These resources include peripherals in the hard processor system (HPS), and custom hardware peripherals implemented within the FPGA in the Cyclone V SoC device.

Contents:

- Getting Started with Linux on the board
- Developing Linux Applications that use FPGA Hardware Devices
- Developing Linux Drivers for FPGA Hardware
- Configuring the FPGA from Linux

Requirements:

- One of the DE-Series development and education boards from Table 1. These boards are described on Intel's FPGA University Program website, and are available from the manufacturer Terasic Technologies.
- Host computer, running either Microsoft Windows (version 10 is recommended) or Linux (Ubuntu, or a similar Linux distribution). The host computer is used for developing software programs that run under Linux on the board
- Ethernet cable, WiFi USB adaptor, and/or Mini-USB cable, for connecting the board to the host computer
- MicroSD card (8 GB or larger)

Optional:

- Intel FPGA SoC Embedded Design Suite (required for Appendix C).
- Intel Quartus Prime Software (required for Appendix D).

DE0-Nano-SoC
DE1-SoC
DE10-Lite
DE10-Standard
DE10-Pro

Table 1. DE-series boards that support the Linux OS.

NOTE: The DE1-SoC is used in this tutorial, but any board from Table 1 will have very similar if not identical instructions

2 Running Linux on the DE-Series Board

Linux is an operating system (OS) that is found in a wide variety of computing products such as personal computers, servers, and mobile devices. Standard distributions of Linux include device drivers for a vast array of hardware devices. In this tutorial we make use of some existing drivers, and also show how the user can make drivers for their own hardware.

2.1 The Cyclone V SoC Device

The supported DE-Series boards features an Intel SoC FPGA, which contains two main components: a *Hard Processor System* (HPS), and a Cyclone V *FPGA*. The HPS contains an ARM Cortex-A9 dual-core processor, which we will use to run Linux, and various peripheral devices such as timers, general-purpose input/output (IO), USB, and Ethernet. The HPS and FPGA are coupled via bridges that allow bidirectional communication. Later in the tutorial, we will show how to write Linux programs that access hardware devices implemented in the FPGA.

2.2 The DE-Series Linux Distribution Images

A number of Linux distributions are available for the aforementioned DE-Series boards. These Linux distributions range from a simple command-line only version to the more full-featured Ubuntu Linux distribution that includes a graphical user interface (GUI). The Linux distributions are provided in the *.img* (image) file format, which can be written onto a microSD card and booted on the board. The custom Linux distribution for the boards in Table 1 can be found from the University Programs website's *Materials* section under *Embedded Linux*.

The Linux distributions contains a number of key features that we will use in this tutorial. First, it provides a GNU Compiler toolchain allowing you to compile C and C++ programs. We will make extensive use of this toolchain to compile programs in Section 3. Another feature is the automatic programming of the FPGA that takes place during the process of booting the OS. The OS programs the FPGA with the University Programs default computer system, which contains IP cores that communicate with the peripheral devices found on the board, such as the switches, LEDs, pushbuttons, VGA, and audio. In Section 3.3, we will show how to write programs that communicate with Parallel IO cores of the *DE1-SoC Computer* to access the LEDs and pushbuttons on the board. The *DE1-SoC Computer* system is described in detail in the document *DE1-SoC Computer with ARM*.

2.3 Preparing the Linux microSD Card

The Linux supporting DE-Series boards are designed to boot Linux from an inserted microSD card. In this section, you will learn how to prepare a Linux microSD card by storing the linux image file onto a microSD card. This section of the tutorial assumes that you have access to a computer with a microSD card reader/writer. To write the image into the microSD card, we will use the free-to-use *Win32 Disk Imager* tool which you can download and install from the Internet. The instructions for using this tool are provided below:

1. Insert a microSD card (8 GB or larger) into your computer's microSD card reader/writer, and then launch the Win32 Disk Imager program.

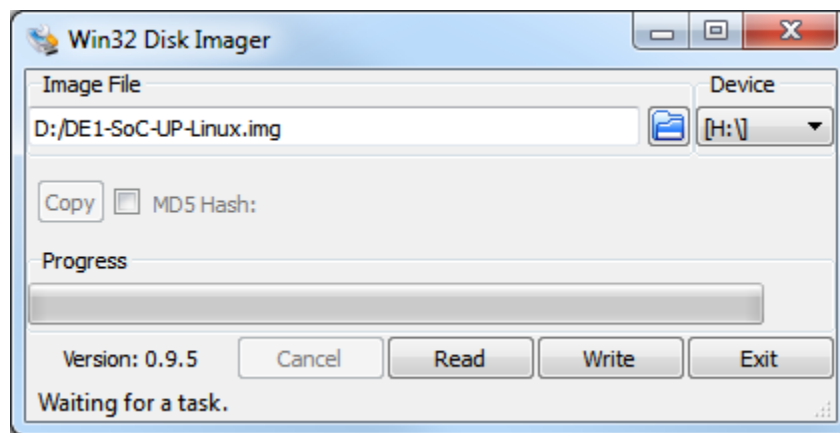


Figure 1. The Win32 Disk Imager program.

2. Select the drive letter corresponding to the microSD card under *Device*, as indicated in Figure 1.
3. Select the desired Linux image under *Image File*, as shown in Figure 1. Again, these images can be found on the University Program's website *Materials* section under the *Embedded Linux subsection*.
4. Click *Write* to write the microSD card. If prompted to confirm the overwrite, press *yes*. Once the writing is complete, you will see the success dialog shown in Figure 2.

2.4 Configuring the Board for use with Linux

First ensure that the board is powered off, and then insert the Linux microSD card into the microSD card slot. Before turning on the board, ensure that the MODE SELECT (MSEL) switches found on the board match the settings shown in Figure 3. These settings configure the Cyclone V SoC chip so as to allow the ARM processor to program the FPGA. It is necessary to have these settings because our Linux image programs the FPGA as part of its boot-up process. We should note that making this change to the MSEL switches does not prevent the FPGA from being programmed using other methods, such as via the Intel Quartus Prime Programming tool.

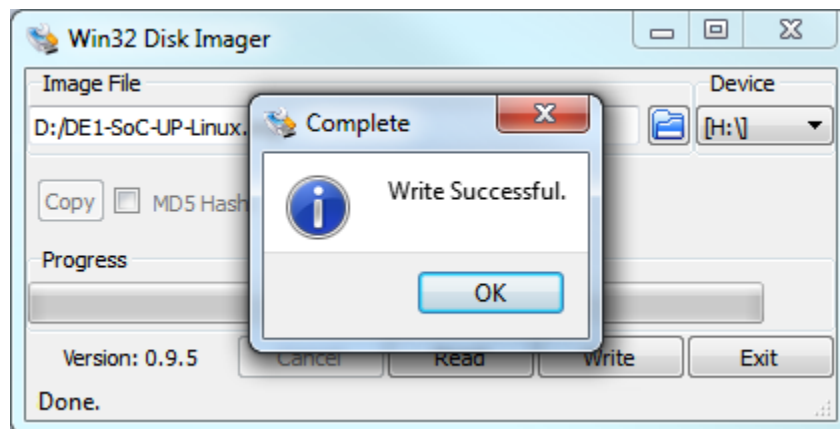


Figure 2. Writing a file to the microSD card using Win32 Disk Imager.

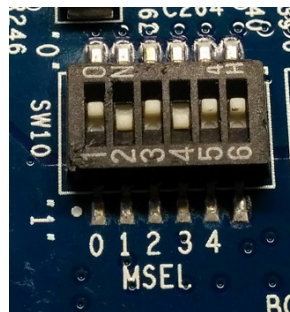


Figure 3. Configuring the MSEL switches of the board.

2.5 Connecting the Board to the Host Computer

Before booting Linux, you should first connect the board to your host computer. There are two main methods of communicating between the board and the host computer: using a *USB cable* to connect to a Linux command-line prompt, or using a *network* to connect to a Linux graphical user interface (GUI). Each method is described below.

2.6 Connecting to the Host Computer using a USB Cable

All DE-Series Linux images have been configured to send and receive text via the Cyclone V HPS's UART. This UART is a device that facilitates serial communication of characters; it is used to send/receive characters to/from the standard Linux streams stdout, stdin, and stderr between the host computer and the UP Linux. On the board, the HPS's UART is attached to a UART-to-USB chip that can be connected to a host computer by using a USB cable. On the host computer, we can use a *terminal* program to display this text. Various *terminal* programs are available via the internet. For this tutorial, we will be using the free-to-use tool `Putty` which is available for both Windows and Linux.

In the following discussion we assume that you have installed Putty on your host computer. If you choose to install and use a different terminal program, then the instructions below would need to be modified accordingly. Connect the UART-to-USB port of the board to your host computer using the mini-USB cable supplied with the board. The UART-to-USB connector can be found immediately next to the microSD card slot. If this is your first time connecting to the UART-to-USB chip, you may have to install its device driver on your host computer. If your host computer's operating system does not automatically install the driver, then you can search for it on the Internet. An appropriate search string is FT232R UART USB Driver, which should locate the driver on a website called ftdichip.com.

2.6.1 Using a Windows Host Computer

On a Windows host computer serial communication devices such as the UART-to-USB are treated as COM ports. Since a host computer may have multiple COM ports, each one is assigned a unique identifying number. The number assigned can be determined by viewing the list of COM ports in *Device Manager*. Figure 4 shows the *Device Manager*'s list of available COM ports on one particular computer. Here, there is only one COM port (the UART-to-USB) which is assigned the number 3 (COM3). If more COM ports were listed, then the UART-to-USB port could be determined by disconnecting and reconnecting the cable to see which COM port disappears, then reappears, in the list.

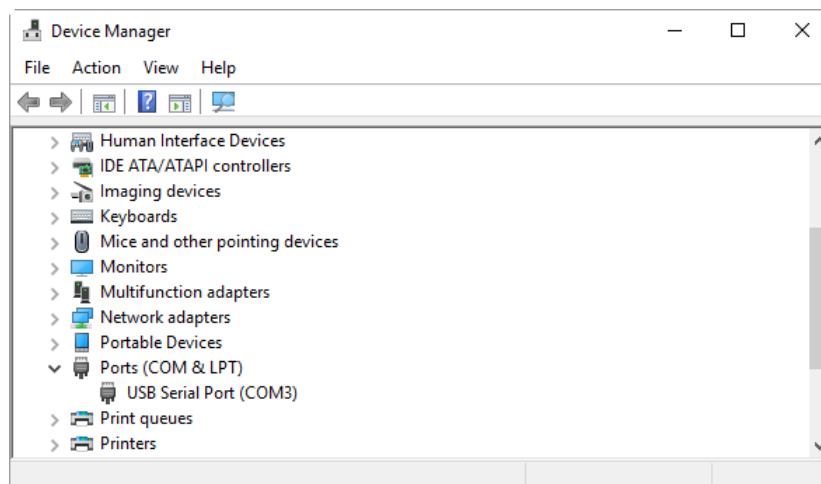


Figure 4. Determining the COM port of the UART-to-USB connection in Device Manager.

2.6.2 Using a Linux Host Computer

On a Linux host computer serial communication devices such as the UART-to-USB are treated as *teletype* (TTY) devices. Since there can be multiple TTY devices connected to the host computer, 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 the figure, you can see that the UART-to-USB chip (the manufacturer's name for this device is *FTDI USB Serial Device converter*) has been assigned the name `ttyUSB0`.

```
kevin@kevin-ThinkPad-T420: ~  
kevin@kevin-ThinkPad-T420:~$ dmesg | grep tty  
[ 0.000000] console [tty0] enabled  
[ 0.537379] 0000:00:16.3: ttyS4 at I/O 0x50b0 (irq = 19, base_baud = 115200)  
is a 16550A  
[ 291.031186] usb 2-1.1: FTDI USB Serial Device converter now attached to ttyUS  
B0  
kevin@kevin-ThinkPad-T420:~$
```

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

2.6.3 Using Putty

Start the Putty program. Now that the serial device (COM port or TTY device) corresponding to the UART-to-USB connection is known, Putty can be configured to connect to it. Figure 6 shows the main window of Putty. In this window, the *Connection type* has been set to *Serial*, and COM3 has been specified in the *Serial line* field.

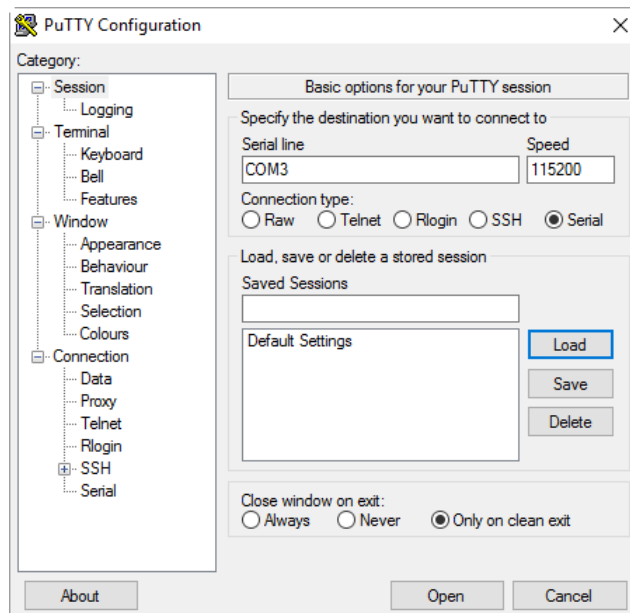


Figure 6. Putty's main window.

Some additional details about the UART-to-USB connection must be entered by selecting the *Serial* panel in the *Category* box on the left side of the window. The *Serial* panel is shown in Figure 7. These settings must match the configuration of the UART. As shown in the figure set the speed (baud rate) to 115200 bits per second, data bits to 8, stop bits to 1, and parity and flow control to *none*.

Once all of the serial-line settings have been entered, press **Open** to start the *Terminal*. Now, turn on the power to the board. You should now see a stream of text in the Putty terminal that shows the status of the Linux boot process, as displayed in Figures 8 and 9. Once Linux has finished booting, you will be logged in to the Linux *command line interface* (CLI) as the *root* user. Being logged in as root means that you that have administrator-level privileges, which allow you to modify settings and execute privileged programs.

In the *Terminal* window press **Enter** on your keyboard to see that the CLI responds. Type a Linux command such as `ls`, which shows a listing of directories and files.

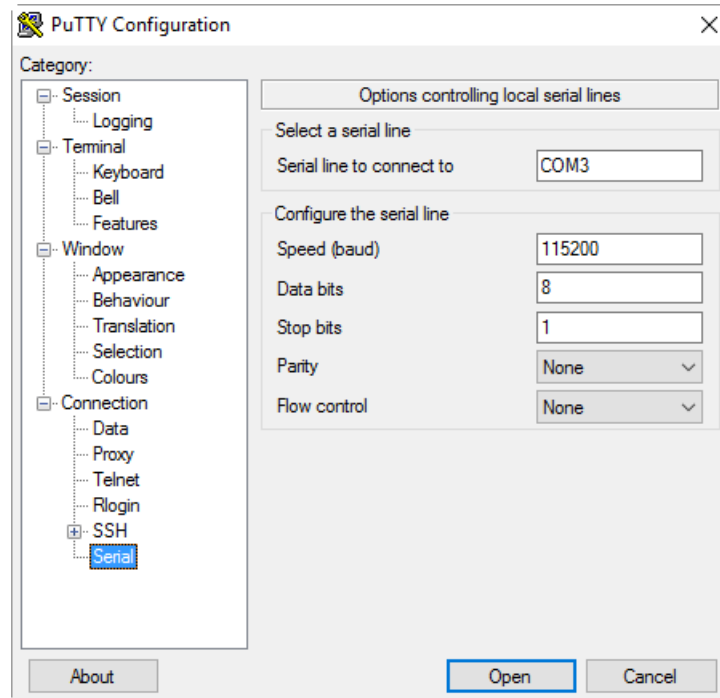
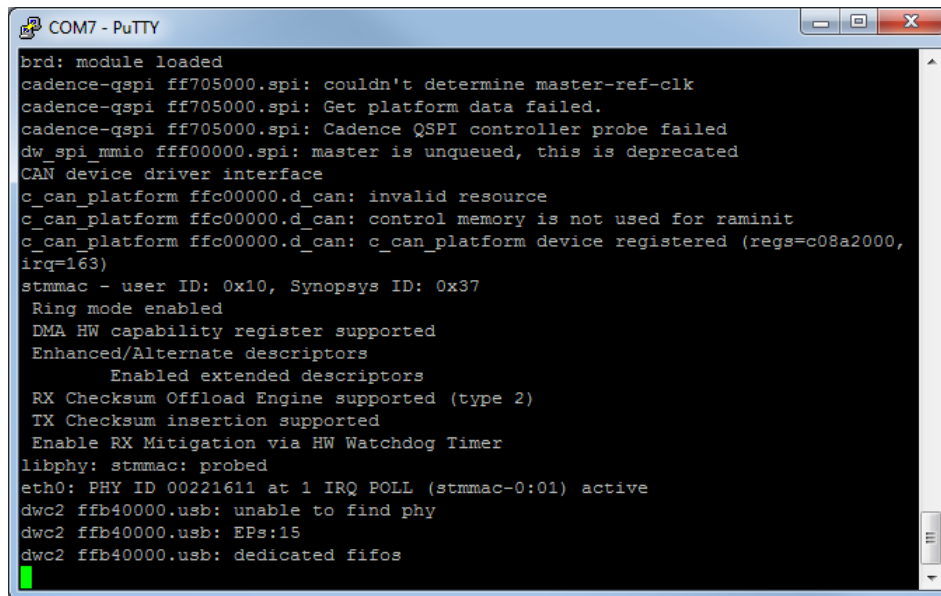


Figure 7. Putty's configuration window for serial communication settings.

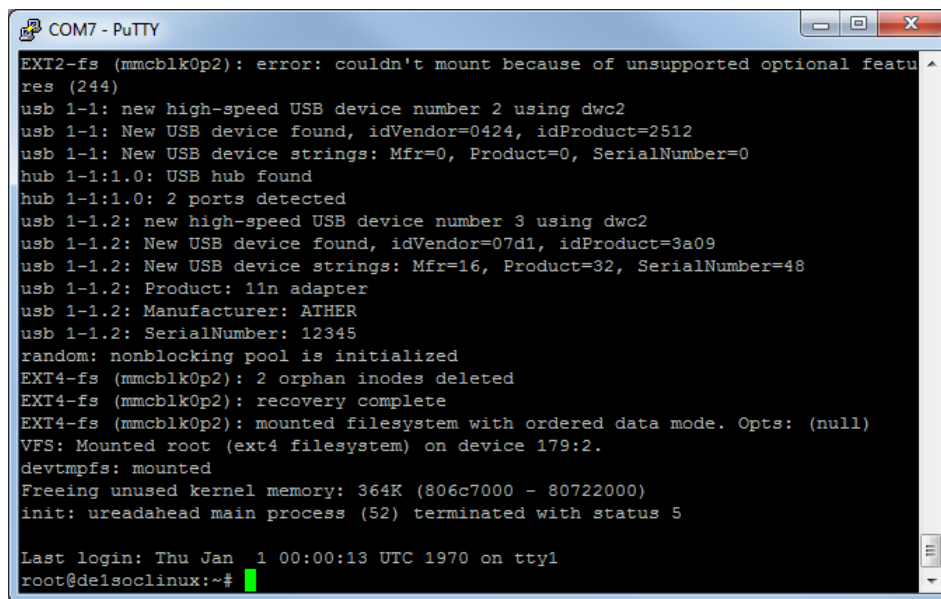


```

COM7 - PuTTY
brd: module loaded
cadence-qspi ff705000.spi: couldn't determine master-ref-clk
cadence-qspi ff705000.spi: Get platform data failed.
cadence-qspi ff705000.spi: Cadence QSPI controller probe failed
dw_spi_mmio fff00000.spi: master is unqueued, this is deprecated
CAN device driver interface
c_can_platform ffc00000.d_can: invalid resource
c_can_platform ffc00000.d_can: control memory is not used for raminit
c_can_platform ffc00000.d_can: c_can_platform device registered (regs=c08a2000,
irq=163)
stmmac - user ID: 0x10, Synopsys ID: 0x37
Ring mode enabled
DMA HW capability register supported
Enhanced/Alternate descriptors
Enabled extended descriptors
RX Checksum Offload Engine supported (type 2)
TX Checksum insertion supported
Enable RX Mitigation via HW Watchdog Timer
libphy: stmmac: probed
eth0: PHY ID 00221611 at 1 IRQ POLL (stmmac-0:01) active
dwc2 ffb40000.usb: unable to find phy
dwc2 ffb40000.usb: EPs:15
dwc2 ffb40000.usb: dedicated fifos

```

Figure 8. Putty terminal displaying text output as the Linux kernel boots.



```

COM7 - PuTTY
EXT2-fs (mmcblk0p2): error: couldn't mount because of unsupported optional featu
res (244)
usb 1-1: new high-speed USB device number 2 using dwc2
usb 1-1: New USB device found, idVendor=0424, idProduct=2512
usb 1-1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
hub 1-1:1.0: USB hub found
hub 1-1:1.0: 2 ports detected
usb 1-1.2: new high-speed USB device number 3 using dwc2
usb 1-1.2: New USB device found, idVendor=07d1, idProduct=3a09
usb 1-1.2: New USB device strings: Mfr=16, Product=32, SerialNumber=48
usb 1-1.2: Product: 11n adapter
usb 1-1.2: Manufacturer: ATHER
usb 1-1.2: SerialNumber: 12345
random: nonblocking pool is initialized
EXT4-fs (mmcblk0p2): 2 orphan inodes deleted
EXT4-fs (mmcblk0p2): recovery complete
EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 179:2.
devtmpfs: mounted
Freeing unused kernel memory: 364K (806c7000 - 80722000)
init: ureadahead main process (52) terminated with status 5
Last login: Thu Jan 1 00:00:13 UTC 1970 on tty1
root@delsoclinux:~#

```

Figure 9. The Linux command line prompt showing the root ('#') logon.

2.7 Connecting to the Host Computer using a Network

The DE-Series Linux images have been configured to include a graphical user interface (GUI), and a virtual network computing (VNC) server. The VNC server transmits a copy of the GUI to a network port, which allows the host

computer to use the GUI via a network connection to the board. The network port used by the VNC server is 5901, and the password for the VNC server is set to *password*.

There are two ways to establish a network connection to the board: using an Ethernet cable, and using a WiFi adapter. Both methods are discussed below.

2.7.1 Connection using an Ethernet Cable

To establish a network connection to your host computer using an Ethernet cable, plug one end of the Ethernet cable into the RJ45 port on the board. The other end of this cable can either be plugged directly into an Ethernet port on the host computer, or plugged into an Ethernet switch on the same network as the host computer. The Linux images are set up to use either of two *IPv4* network addresses via Ethernet: 192.168.0.123 and 192.168.1.123. For the host computer to connect to the board, the host computer's network address must be on the same *subnet*. This means that the host computer's network address must be of the form 192.168.0.xxx or 192.168.1.xxx. The steps required to complete the Ethernet connection to the VNC server are described below.

You first need to determine your host computer's IP address. If you are using Windows on the host computer, open a Command (CMD) prompt window and execute `ipconfig`. In the output produced by this command look for the computer's *IPv4 Address*. If you are running Linux on your host computer, open a Terminal window and run the command `ifconfig`. In the output look for an *inet addr* that is associated with an Ethernet port, such as *eth0*.

If your host address is on subnet 192.168.0 or 192.168.1, then skip to Section 2.7.3. Otherwise, you need to change either the IP address of your host computer, or the IP address of the board. If your host computer is currently connected to the Internet, and you wish to maintain this Internet connection, then the best option is to change the IP address of the board so that it uses the same subnet as the host computer. But if you do not need to access the Internet on the host computer, then you may wish to instead change the IP address of your host computer to use the same subnet as the board.

The procedures for changing the IP address of the host computer or board are described below.

Changing the IP address of your Host Computer

Note that if you are currently connected to the Internet on your host computer, and wish to maintain this connection, then you probably do not want to change your host computer's IP address. This is because your host computer's IP address would be set to allow it to communicate with your Internet modem or router. But if you are not using the Internet on the host computer, then the procedure below may be used to change its IP address.

If you are using Windows, the IP address of the host computer can be changed by using the Windows *Control Panel*. In the *Control Panel*, open the *Network and Sharing Center* item. Click on *Change adapter settings*, then right-click on your Ethernet adapter and open the *Properties* dialog. Highlight the *Internet Protocol Version 4 (TCP/IPv4)* item and click *Properties*. In the *General* tab click *Use the following IP address*. In the *IP address* field enter 192.168.0.xxx (or 192.168.1.xxx), where xxx is a number of your choosing (that is not already being used in the subnet). In the *Subnet mask* field enter 255.255.255.0. Leave the *Default gateway* field blank.

If you are using Linux on the host computer, the IP address can be changed by using the `ifconfig` command. If your Ethernet adapter were called *eth0*, then the command would be `ifconfig eth0 192.168.0.xxx` (or

192.168.1.xxx), where xxx is a number of your choosing (that is not already being used in the subnet). If your Ethernet adapter is not called *eth0*, then replace this field with the actual name of your Ethernet port.

Once you have established the correct IP address, you can connect to the VNC server as described in Section 2.7.3.

Changing the IP address of the Board

To change the IP address of the board you have to first connect your host computer to the board via a USB cable, as described in section 2.6. Then, using a *Terminal* window connected to Linux on the board execute the *ifconfig* command. For example, if your host computer's IP address were 169.254.245.156, then you would use the command *ifconfig eth0 169.254.245.xxx*, where xxx is a number of your choosing (that is not already being used in the subnet).

Once you have established the correct IP address, you can connect to the VNC server as described in Section 2.7.3.

2.7.2 Connecting to the Host Computer using a WiFi Adapter

To make a network connection to the board using a WiFi adapter, you first have to connect your host computer to the board via a USB cable, as described in section 2.6. Then, you can use a Terminal window connected to Linux on the board to connect to the desired WiFi network.

The University Program's Linux images support a variety of USB WiFi adapters. At the time of writing this tutorial WiFi adapters supported by Linux kernel version 3.18 have been tested. Other WiFi adapters may also be usable, but drivers may need to be manually installed.

Plug your WiFi adapter into a USB port on the board. To join a desired WiFi network, you can run the following script: *connect_wpa <ssid> <password>*. This script can be found in the directory */home/root/misc* in the Linux filesystem. The board should become connected to your WiFi network after a few moments.

Instead of using the *connect_wpa* script, it is possible to run the Linux commands included in the script manually. First, using the Terminal window connected to Linux on the board, create an ASCII text file in a directory of your choosing. Give the file a name ending in *.conf*, such as *mywifi.conf*. This file has to contain the lines

```
network={
    ssid="<ssid>"
    psk="<password>"
}
```

Note that the first character on each of the second and third lines is a *tab* character. Now, run the following Linux commands:

```
stop network-manager
wpa_supplicant -B -iwlan0 -c./mywifi.conf -Dnl80211
dhclient wlan0
```

Note that the wireless interface on your board might not be *wlan0*. To determine the correct name, use the Linux command *iwconfig*. You can check the IP address assigned by the WiFi router using the Linux *ifconfig* command. Then, you can then use this IP address, as described in Section 2.7.3, to connect to the VNC server from

a host computer on the same WiFi network.

2.7.3 Using the VNC Server

After you have set up a network connection between your host computer and the board, you can use a *VNC Viewer* application on your host computer to connect to the Linux GUI. For this tutorial we will be using the *RealVNC Viewer* that is available for no charge for Windows computers. For Linux host computers you can use a VNC Viewer such as the *Remmina* application that is included with some Linux distributions.

Figure 10 shows the opening dialog of the *RealVNC Viewer* application. The VNC Server address is specified assuming that the default IP address `192.168.0.123` is being used, and port `5901` is specified as required. Clicking on the *Connect* button opens another dialog that prompts for the password. Providing the password (which is just set to *password* in our case) opens the RealVNC window shown in Figure 11. In the figure we have opened the *Terminal* command prompt window inside the VNC Viewer and typed the Linux command `pwd`.

The VNC Server on the Computer supports four different screen sizes. Using a *Terminal* command prompt window in the VNC Viewer the screen size can be changed with the `xrandr` command. You can type `xrandr -s 0` to select the smallest screen size, and `xrandr -s 3` to choose the largest size.

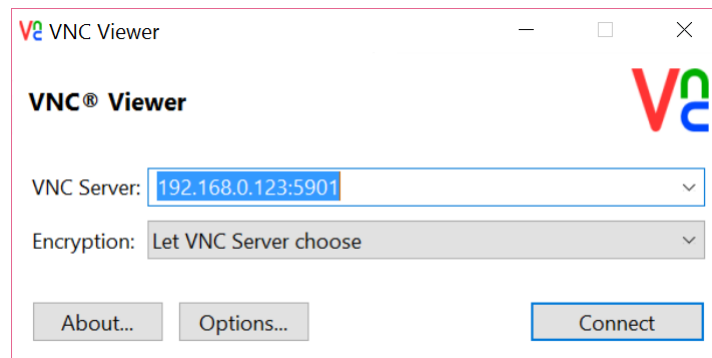


Figure 10. The RealVNC opening dialog.

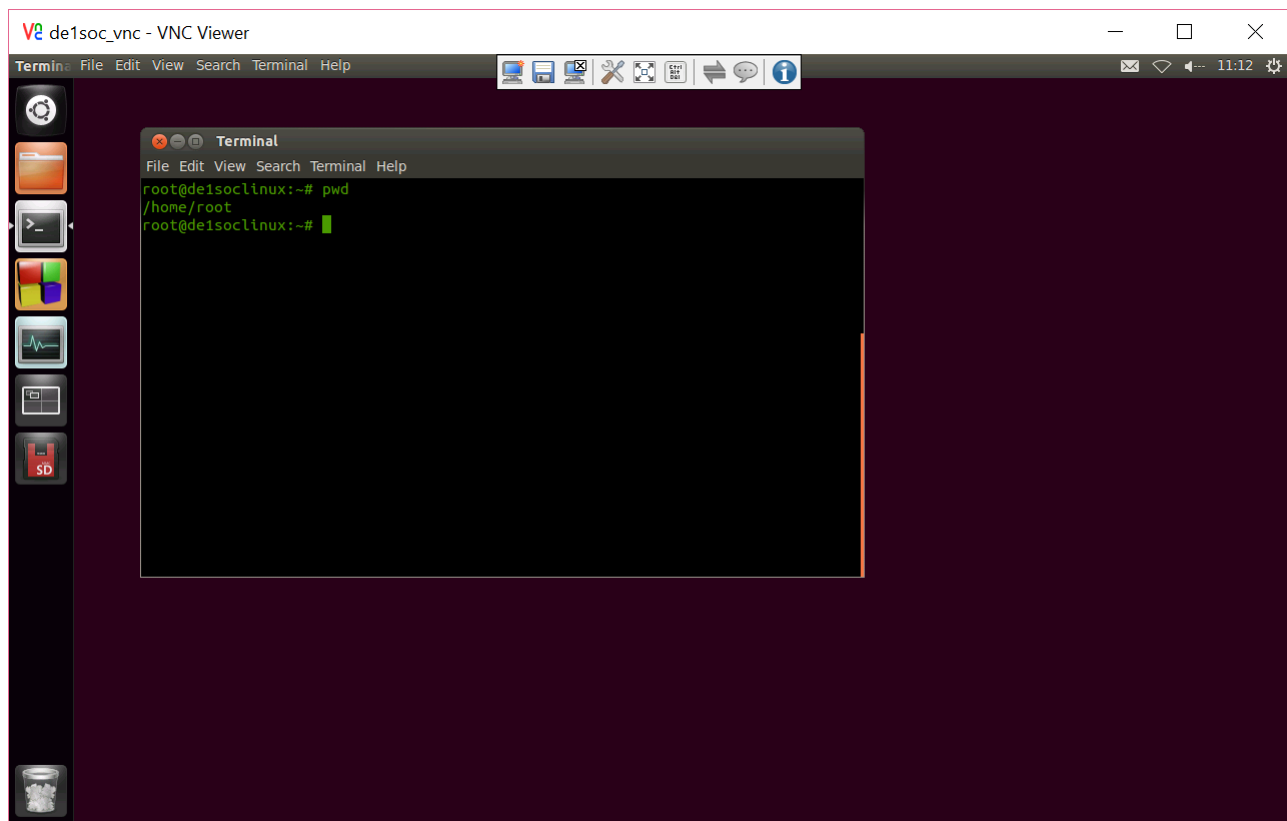


Figure 11. The main RealVNC window.

The Linux images include many application programs. It provides several text editors, including *gedit*, *Emacs*, *vim*, and *gvim*. It also includes the *Code::Blocks* integrated development environment. This tool provides a source-level debugger that can be used to develop applications programs. In Appendix A we provide a short tutorial that shows how to use *Code::Blocks* to develop C programs that run on the ARM processor under Linux.

2.7.4 Transferring Files to/from the Host Computer

The Linux images include an *FTP* Server that can be used to transfer files between the host computer and the board computer. The FTP server uses the secure FTP (SFTP) protocol and uses *Port 22* of the network connection. The login *Username* for the FTP Server is *root*, and the password is *password*. An easy-to-use FTP client for both Windows and Linux host computers is *FileZilla*, available from <https://filezilla-project.org/>.

2.7.5 Accessing the Internet

After connecting the board to a network, it is possible to provide Internet access to Linux applications. For example, the Linux images include the *Mozilla Firefox* browser, which can be used to browse web pages. If you are connected to the network using WiFi as discussed in Section 2.7.2, then the Internet access provided by your WiFi router is already enabled. But if you have connected using an Ethernet cable, then the following commands have to be used

to enable the Internet access provided by your router:

```
ifconfig eth0 0.0.0.0 0.0.0.0
dhclient eth0
```

These commands set up the *eth0* port as a DHCP client of the router. The *eth0* port will obtain an automatically-assigned IP address that allows Internet access. You can use this IP address to connect to the VNC server as described in Section 2.7.3, and can run Linux applications that access the Internet.

3 Developing Linux Programs for the Board

In this section you will learn how to develop programs that can run under Linux on the board (specifically the DE1-SoC board). There are two options for developing a Linux program for the board. The first is to write and compile code using either the command-line or GUI (VNC) interface of the Linux running on the board. This approach is called *native compilation*, and is described in Section 3.1. Native compilation is the primary method used in this tutorial. The second option is to write and compile your program on a host computer, and then transfer the resulting executable onto the Linux filesystem (microSD card). This approach is called *cross compilation*, and is described briefly in Appendix C.

3.1 Native Compilation on the 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, we will be natively compiling a program through the Linux command-line interface, using its built-in compilation toolchain.

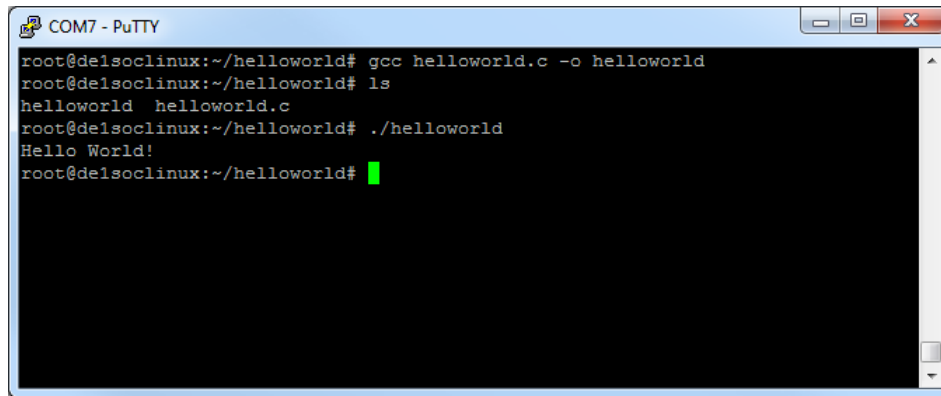
To demonstrate native compilation, we will compile a simple "hello world" program. The code for this program is shown in Figure 12. You can also find the code in */home/root/helloworld/helloworld.c* of the Linux filesystem.

```
1  #include <stdio.h>
2
3  int main(void) {
4
5      printf("Hello World!\n");
6
7      return 0;
8  }
```

Figure 12. The helloworld program

You can compile code using the Linux command-line interface. If you are using a USB cable to connect to the board, as discussed in Section 2.6, then use the *Putty* tool to open a *Terminal* window. If you are using the VNC Viewer, as described in Section 2.7, then open a *Terminal* window in the GUI. In your *Terminal* window change the working directory to */home/root/helloworld*. Compile the program using the command `gcc helloworld.c -o helloworld`, as shown in Figure 13. The `gcc` command invokes the *GNU C Compiler*, which is an open-source compiler that is widely used to compile Linux programs. 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 is complete, we can run the program by typing `./helloworld`. The program outputs the message "Hello World!" then exits, as shown in Figure 13.



```
COM7 - PuTTY
root@de1soclinux:~/helloworld# gcc helloworld.c -o helloworld
root@de1soclinux:~/helloworld# ls
helloworld  helloworld.c
root@de1soclinux:~/helloworld# ./helloworld
Hello World!
root@de1soclinux:~/helloworld#
```

Figure 13. Compiling and executing the *helloworld* program through the command line.

3.2 Accessing Hardware Devices in the FPGA from a Linux Program

Programs running on the ARM processor in the DE-Series Linux OS can access hardware peripherals that are implemented in the FPGA. The ARM processor can access the FPGA by using either the *HPS-to-FPGA bridge* 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 “bare metal” 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. Examples of software programs that access memory-mapped peripherals in the FPGA can be found on the Intel FPGA University Program website in the document *Intel FPGA Monitor Program Tutorial for ARM*. But 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 then 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 special 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 in the FPGA.

3.3 Example Program that uses an FPGA Hardware Device

In this section, we describe an example of code in the C language that uses a hardware device in the FPGA. The application program alters the state of the red LEDs on the DE1-SoC board. Recall that the Linux distribution automatically downloads the circuit that implements the *DE1-SoC Computer* system into the FPGA during the boot process. The *DE1-SoC Computer* includes a parallel port that is connected to the red LEDs on the board. This parallel port is attached to the *lwahps2fpga* 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 $0xFF200000 + \text{offset}$. 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. A diagram showing how the red LEDs are connected to the parallel port is shown in Figure 14.

The code for the application program is shown in Figure 15. Each time this program is executed, the value displayed on the red LEDs is incremented by one. The example code can be found on the Linux microSD card in the file `/home/root/increment_leds/increment_leds.c`. You can compile the code using a command such as `gcc -Wall increment_leds.c -o increment_leds`. Some important lines in the code are described below.

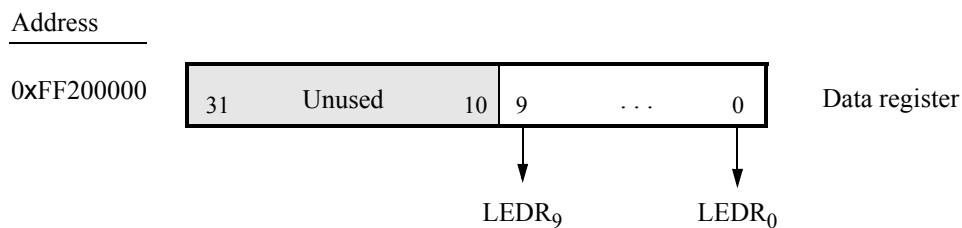


Figure 14. The LED parallel port.

```

1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <sys/mman.h>
4  #include "../address_map_arm.h"
5
6  /* Prototypes for functions used to access physical memory addresses */
7  int open_physical (int);
8  void * map_physical (int, unsigned int, unsigned int);
9  void close_physical (int);
10 int unmap_physical (void *, unsigned int);
11
12 /* This program increments the contents of the red LED parallel port */
13 int main(void)
14 {
15     volatile int * LEDR_ptr;    // virtual address pointer to red LEDs
16     int fd = -1;                // used to open /dev/mem
17     void *LW_virtual;           // physical addresses for light-weight bridge
18
19     // Create virtual memory access to the FPGA light-weight bridge
20     if ((fd = open_physical (fd)) == -1)
21         return (-1);
22     if ((LW_virtual = map_physical (fd, LW_BRIDGE_BASE, LW_BRIDGE_SPAN)) ==
        NULL)
23         return (-1);
24
25     // Set virtual address pointer to I/O port
26     LEDR_ptr = (unsigned int *) (LW_virtual + LEDR_BASE);
27     *LEDR_ptr = *LEDR_ptr + 1; // Add 1 to the I/O register
28
29     unmap_physical (LW_virtual, LW_BRIDGE_SPAN);
30     close_physical (fd);
31     return 0;
32 }

```

Figure 15. C-code for the *increment_leds* program

- Lines 2-3 include the `fcntl.h` and `sys/mman.h` header files, which are needed to use the `/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. The contents of this file are listed in Appendix B.
- Lines 7-10 provide prototype declarations for functions that are used to access physical memory. These functions are listed in Figure 16. The functions `open_physical` and `close_physical` are used to open and 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, along with the address information given in Appendix B.

- Line 20 opens the file `/dev/mem`
- Line 22 maps a part of the `/dev/mem` file into memory. It maps a portion that starts at the base address of `lwyps2fpga`, specified in the code as `LW_BRIDGE_BASE`, and spans `LW_BRIDGE_SPAN` bytes. Appendix B gives the values of `LW_BRIDGE_BASE` and `LW_BRIDGE_SPAN`. 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 that an access to `LW_virtual + offset` will access the physical address `0xFF200000 + offset`.
- Line 26 calculates the virtual address that maps to the LED port. This is done by adding the address offset of the port, `LEDR_BASE`, to `LW_virtual`.
- Line 27 reads the *data* register of the LED port, increments the value by one, then writes the incremented value back to the register.
- Lines 29-30 unmap and close the `/dev/mem` file

```

1  /* Open /dev/mem to give access to physical addresses */
2  int open_physical (int fd)
3  {
4      if (fd == -1) // check if already open
5          if ((fd = open( "/dev/mem", (O_RDWR | O_SYNC))) == -1)
6              {
7                  printf ("ERROR: could not open \"/dev/mem\"...\n");
8                  return (-1);
9              }
10     return fd;
11 }
12
13 /* Close /dev/mem to give access to physical addresses */
14 void close_physical (int fd)
15 {
16     close (fd);
17 }
18
19 /* Establish a virtual address mapping for the physical addresses starting
20  * at base and extending by span bytes */
21 void* map_physical(int fd, unsigned int base, unsigned int span)
22 {
23     void *virtual_base;
24     // Get a mapping from physical addresses to virtual addresses
25     virtual_base = mmap (NULL, span, (PROT_READ | PROT_WRITE), MAP_SHARED,
26                          fd, base);
27     if (virtual_base == MAP_FAILED)
28     {
29         printf ("ERROR: mmap() failed...\n");
30         close (fd);
31         return (NULL);
32     }
33     return virtual_base;

```

Figure 16. Functions for managing physical memory addresses (Part a).

```

1  /* Close the previously-opened virtual address mapping */
2  int unmap_physical(void * virtual_base, unsigned int span)
3  {
4      if (munmap (virtual_base, span) != 0)
5      {
6          printf ("ERROR: munmap() failed...\n");
7          return (-1);
8      }
9      return 0;
10 }

```

Figure 16. Functions for managing physical memory addresses (Part b).

3.4 Device Drivers

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; making pre-compiled device drivers that are distributed with the Linux kernel is beyond the scope of this tutorial.

The kernel module described in this section uses the pushbutton KEY port in the DE1-SoC Computer. To make the example more interesting, we use ARM processor interrupts to handle KEY presses. A diagram of the pushbutton KEY port is shown in Figure 17. There is a *Data* register that reflects which KEY(s) are pressed at a given time. For example, if *KEY*₀ is currently being pressed, then bit 0 of the data register will be 1, otherwise 0. The *Edgecapture* register can be used to check if a *KEY* has been pressed since last examined, even if it has since been released. If, for example, *KEY*₀ is pressed, then bit 0 of the *Edgecapture* register becomes 1. This bit remains 1 even if *KEY*₀ is released. To reset the bit to 0, the ARM processor has to explicitly write the value 1 into this bit-position of the *Edgecapture* register. The KEY port can send interrupts to the ARM processor. Interrupts can be enabled for each *KEY* separately, using the *Interruptmask* register. An interrupt for a *KEY* is enabled by setting the corresponding bit in the *Interruptmask* register to 1.

Address	31	30	...	4	3	2	1	0	
0xFF200050	Unused				KEY ₃₋₀				Data register
Unused	Unused								
0xFF200058	Unused				Mask bits				Interruptmask register
0xFF20005C	Unused				Edge bits				Edgecapture register

Figure 17. The pushbutton KEY parallel port.

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 accommodate 256 interrupt request (IRQ) lines

IRQ₀ to IRQ₂₅₅. A total of 64 of the lines (IRQ₇₂ - IRQ₁₃₅) are reserved for interrupts originating from hardware devices implemented inside the FPGA. In the DE1-SoC Computer the pushbutton KEY port is connected to interrupt line IRQ₇₃. This 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 a high-level interface provided by the OS 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`.

3.4.1 The Pushbutton Interrupt Handler Kernel Module

The code for our kernel module is shown in Figure 18. Lines 1-5 in this code include various header files that are needed for our kernel module. Line 6 include a file that specifies addresses, and line 7 includes a file that lists all FPGA interrupts in the DE1-SoC Computer. These files are provided in Appendix B. Kernel modules, unlike regular C programs, do not have a `main` function. Instead, kernel modules have an `init` function which is executed when the module is inserted into the kernel, and an `exit` function which is executed if 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)`. In this function, line 23 makes a *system call* to the function `ioremap_cache(base_address, span)`, which is part of the Linux kernel. This function allows the kernel module to access physical memory addresses. The `ioremap_cache` function has a similar purpose as the `mmap` function that we discussed in Section 3.3. Kernel modules are not allowed to call the `mmap` function, and instead have to use the `ioremap_cache` function. This function returns a virtual address that can be used to access physical memory starting at *base_address* and extending *span* bytes.

Line 25 of the code sets up a virtual address pointer for the LED parallel port, and line 26 initializes the value of this port to 0x200, which turns on the leftmost LED (as a visual indication that the module has been inserted). The code then configures the pushbutton port so that it will generate an interrupt when a button is pressed. Finally, line 33 calls `request_irq(...)` to register our `irq_handler(...)` to handle pushbutton interrupts. 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 KEY 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 the like. A device driver module that does not use interrupts would still look similar to the code in Figure 18, but without the interrupt-specific code like `irq_handler` and `free_irq`. The `exit` function in our module is `cleanup_pushbutton_handler(void)`. It sets LEDs to 0x0, turning them off, and de-registers the pushbutton `irq` handler by calling the `free_irq(...)` function.

```

1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/init.h>
4  #include <linux/interrupt.h>
5  #include <asm/io.h>
6  #include "../address_map_arm.h"
7  #include "../interrupt_ID.h"
8
9  void * LW_virtual;           // Lightweight bridge base address
10 volatile int *LEDR_ptr, *KEY_ptr; // virtual addresses
11
12 irq_handler_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
13 {
14     *LEDR_ptr = *LEDR_ptr + 1;
15     // Clear the Edgecapture register (clears current interrupt)
16     *(KEY_ptr + 3) = 0xF;
17     return (irq_handler_t) IRQ_HANDLED;
18 }
19 static int __init initialize_pushbutton_handler(void)
20 {
21     int value;
22     // generate a virtual address for the FPGA lightweight bridge
23     LW_virtual = ioremap_nocache (LW_BRIDGE_BASE, LW_BRIDGE_SPAN);
24
25     LEDR_ptr = LW_virtual + LEDR_BASE; // virtual address for LEDR port
26     *LEDR_ptr = 0x200;                // turn on the leftmost light
27
28     KEY_ptr = LW_virtual + KEY_BASE;   // virtual address for KEY port
29     *(KEY_ptr + 3) = 0xF; // Clear the Edgecapture register
30     *(KEY_ptr + 2) = 0xF; // Enable IRQ generation for the 4 buttons
31
32     // Register the interrupt handler.
33     value = request_irq (KEYS_IRQ, (irq_handler_t) irq_handler, IRQF_SHARED,
34         "pushbutton_irq_handler", (void *) (irq_handler));
35     return value;
36 }
37 static void __exit cleanup_pushbutton_handler(void)
38 {
39     iounmap (LW_virtual);
40     *LEDR_ptr = 0; // Turn off LEDs and de-register irq handler
41     free_irq (KEYS_IRQ, (void*) irq_handler);
42 }
43 module_init(initialize_pushbutton_handler);
44 module_exit(cleanup_pushbutton_handler);

```

Figure 18. C-code for the pushbutton interrupt handler kernel module

3.4.2 Compiling the Kernel Module

The kernel module 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 19. The first line, `obj-m += <module_name>.o`, specifies the name of the kernel module that is to be built (our kernel module will as a result be named `pushbutton_irq_handler`). This line also tells the build system to look for the kernel module code in `<module_name>.c`, and generate the kernel object file `<module_name>.ko` at the end of the compilation.

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

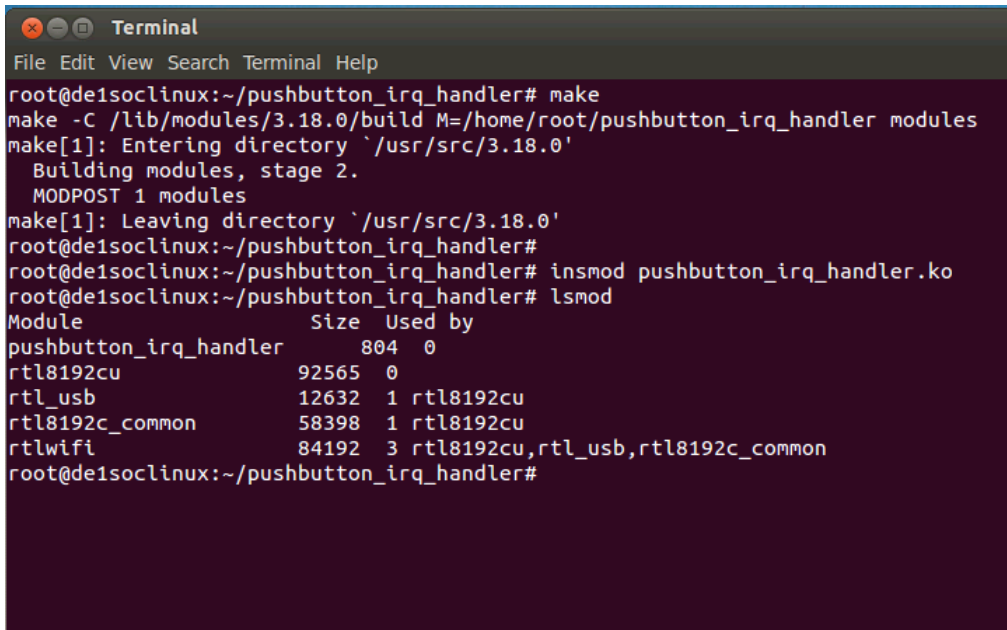
```
1  obj-m += pushbutton_irq_handler.o
2
3  all:
4      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6  clean:
7      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Figure 19. Kernel module makefile

3.4.3 Running the Kernel Module

A kernel module is executed by *inserting* it into the Linux kernel using the command `insmod <module_name.ko>`. Insert the kernel module you compiled above by using the command `insmod pushbutton_irq_handler.ko`, as shown in Figure 20. You can use the command `lsmod` to confirm that your module 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 `IRQ73`, and confirm that the value displayed on the LEDs increments by one.

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

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
root@de1soclinux:~/pushbutton_irq_handler# make
make -C /lib/modules/3.18.0/build M=/home/root/pushbutton_irq_handler modules
make[1]: Entering directory `/usr/src/3.18.0'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory `/usr/src/3.18.0'
root@de1soclinux:~/pushbutton_irq_handler#
root@de1soclinux:~/pushbutton_irq_handler# insmod pushbutton_irq_handler.ko
root@de1soclinux:~/pushbutton_irq_handler# lsmod
Module                Size  Used by
pushbutton_irq_handler    804  0
rtl8192cu              92565  0
rtl_usb                12632  1 rtl8192cu
rtl8192c_common         58398  1 rtl8192cu
rtlwifi                84192  3 rtl8192cu,rtl_usb,rtl8192c_common
root@de1soclinux:~/pushbutton_irq_handler#
```

Figure 20. Inserting and removing the kernel module

Appendix A Using Code::Blocks

If you are using a network connection to your Board's Computer, as described in Section 2.7, then you can make use of the Code::Blocks tool for developing and debugging application programs. In this appendix we provide a simple example that shows how to create a Code::Blocks *project* on the Board's Computer. We will show how to build a project using the *increment_leds* example that we discussed in Section 3.3.

Open the Code::Blocks tool by clicking on its icon, which looks like four colored cubes. The main window of Code::Blocks is displayed in Figure 21. In this window, click on **Create a new project** to begin using the tool. In the window that opens click on the **Empty project** item, and then click on the **Go** button.

In the *Empty project* dialog, shown in Figure 22, type a title for the project, such as *increment_leds*. Use the **...** button to navigate to, and select, the folder that contains the *increment_leds* source code. Give the project a name like *increment_leds*. Make sure that the **Resulting filename** item shows the proper name and path to the project. Click on the **Next** button to reach a second *Empty project* dialog. Then, click **Finish** to return to the main Code::Blocks window.

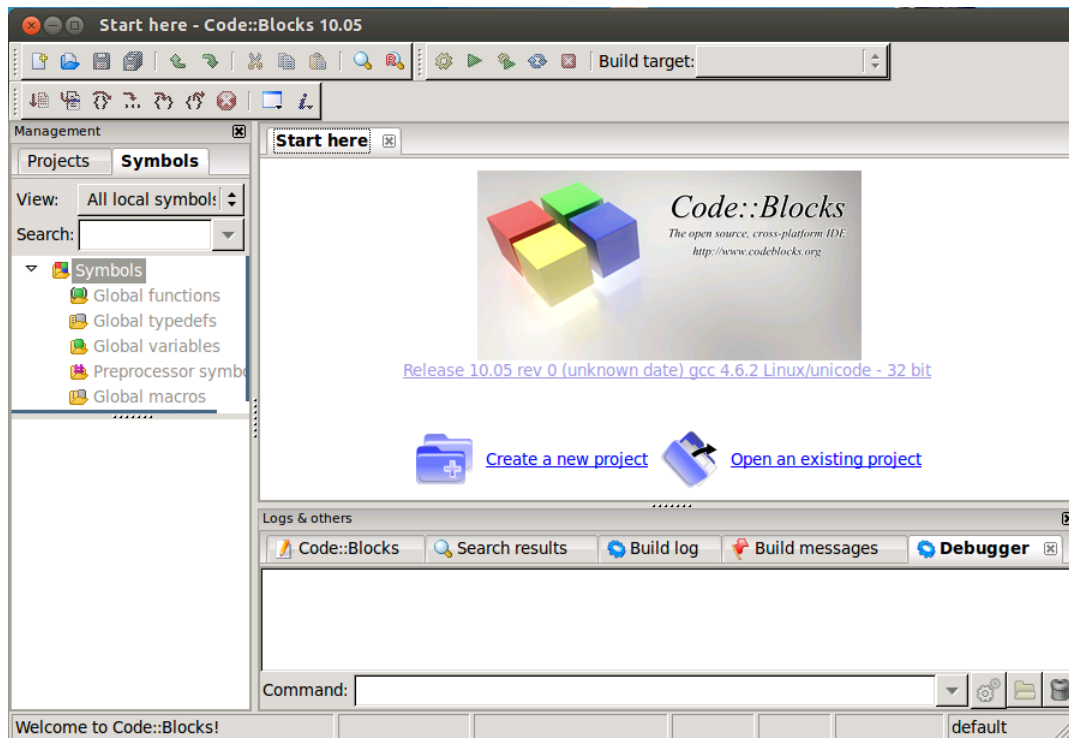


Figure 21. The main Code::Blocks window.

As indicated in Figure 23 right-click the *increment_leds* name under **Workspace**, and then select **Add files...**. Select the *increment_leds.c* source-code file, as illustrated in Figure 24, and then click the **Open** button. In the dialog that opens, illustrated in Figure 25, select **OK**. Now you can open the *increment_leds* item under **Workspace**, then open the **Sources** sub-menu, and double-click to open the *increment_leds.c* file inside the Code::Blocks window. You can change the size of the displayed text by holding down the **CTRL** key and rolling the mouse wheel.

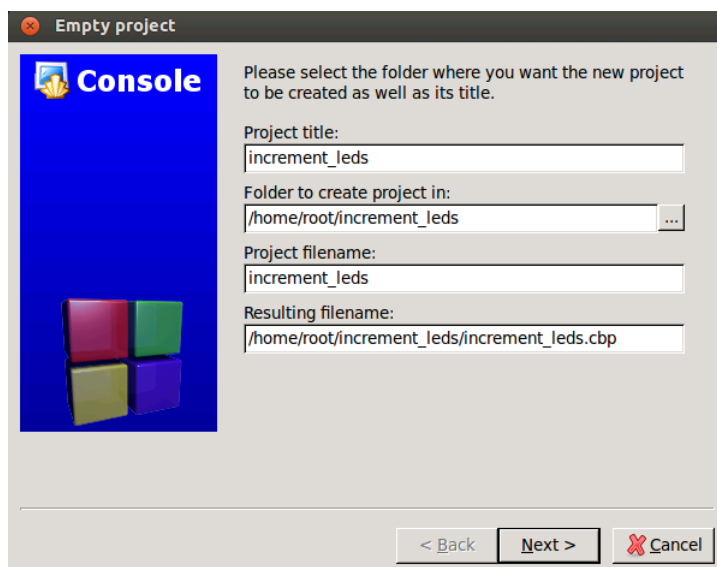


Figure 22. The Empty project dialog.

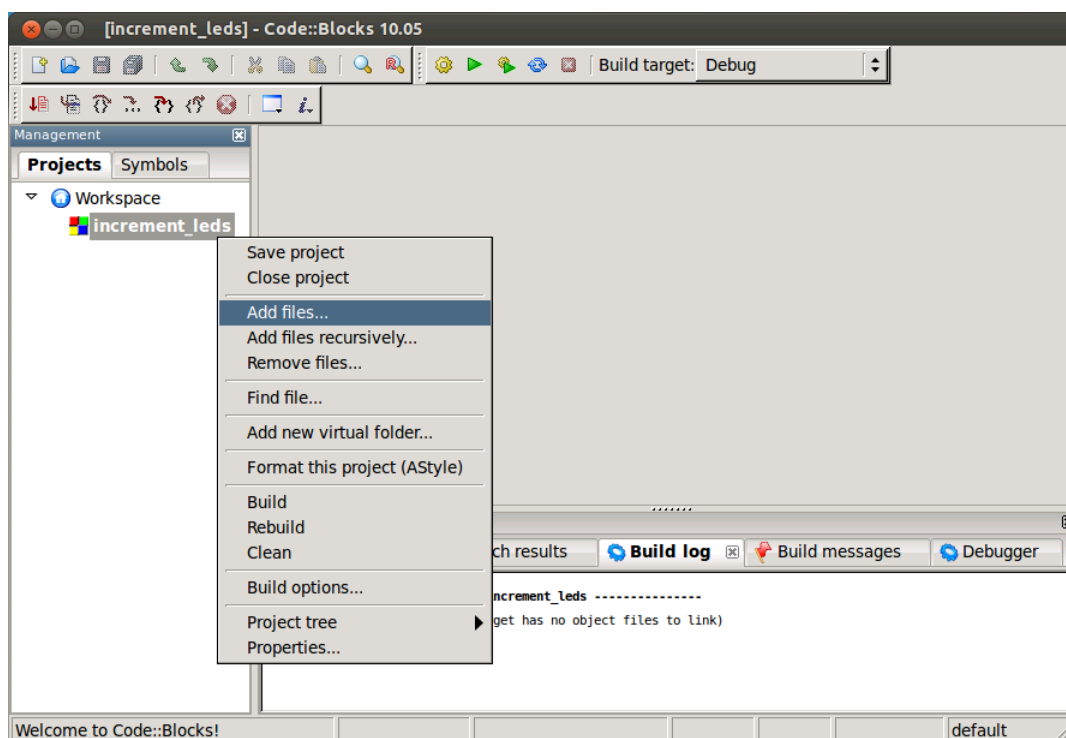


Figure 23. Adding source-code files to the project.

Click to the right of the line of code that calls the function `open_physical`, as shown in Figure 26, and set a *breakpoint*. The breakpoint is indicated by a red circle.

Now start the debugger by selecting the command `Debug | Start`, as indicated in in Figure 27. (Note that the main menu commands for `Code :: Blocks` are provided at the top of the Linux desktop, and not in the border of the `Code :: Blocks` window.) The debugger will start running the program and it will stop when the code reaches the breakpoint. Figure 28 shows the debugger window after reaching the breakpoint. If the CPU Registers window is not visible, it can be opened by selecting the command `Debug | Debugging windows | CPU Registers`. This window shows the current contents of the ARM processor general-purpose registers.

The debugger can display the values of variables used in your program, as illustrated in Figure 29. Expand the `Local variables` item in the `Watches` window to see the variables that currently exist in the program. Execute a few more lines of code until the value of the `LEDR_ptr` variable is initialized by the program, as displayed in the figure. To execute a line of code use the command `Debug | Next line`. This command is available in the main *Debug* menu, via the short-cut keyboard key `F7`, or by clicking on its icon in the *Debug toolbar*. If this toolbar is not open when the debugger is running, it can be opened by using the command `View | Toolbars | Debugger`.

More complete information about using `Code :: Blocks` can be found by searching for documentation and tutorials on the Internet.

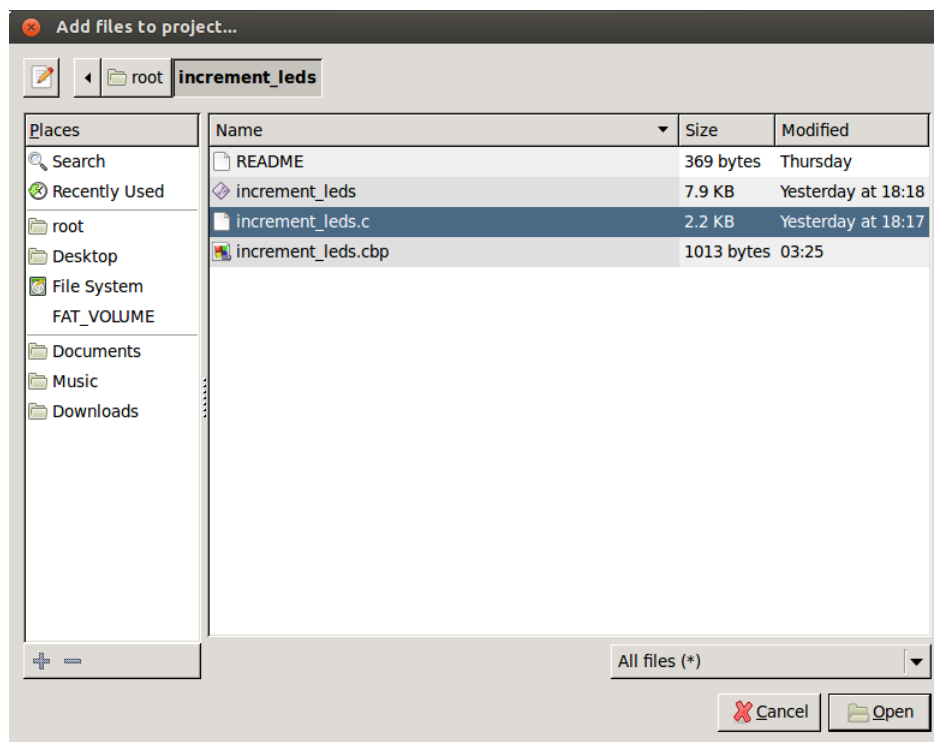


Figure 24. Selecting the *increment_leds.c* file.

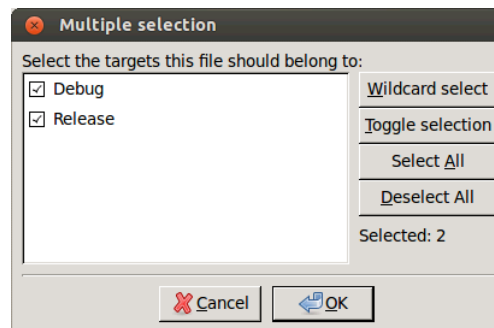


Figure 25. Selecting build targets.

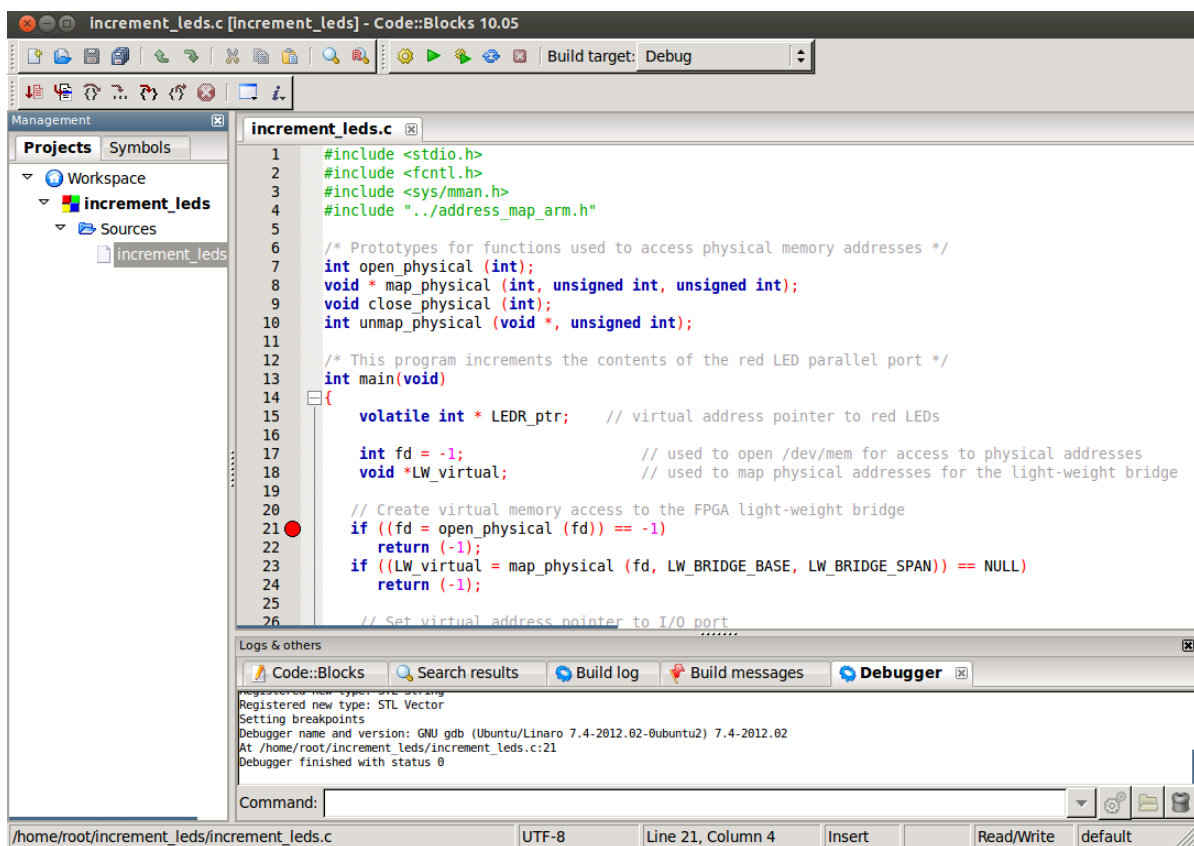


Figure 26. Setting a breakpoint.

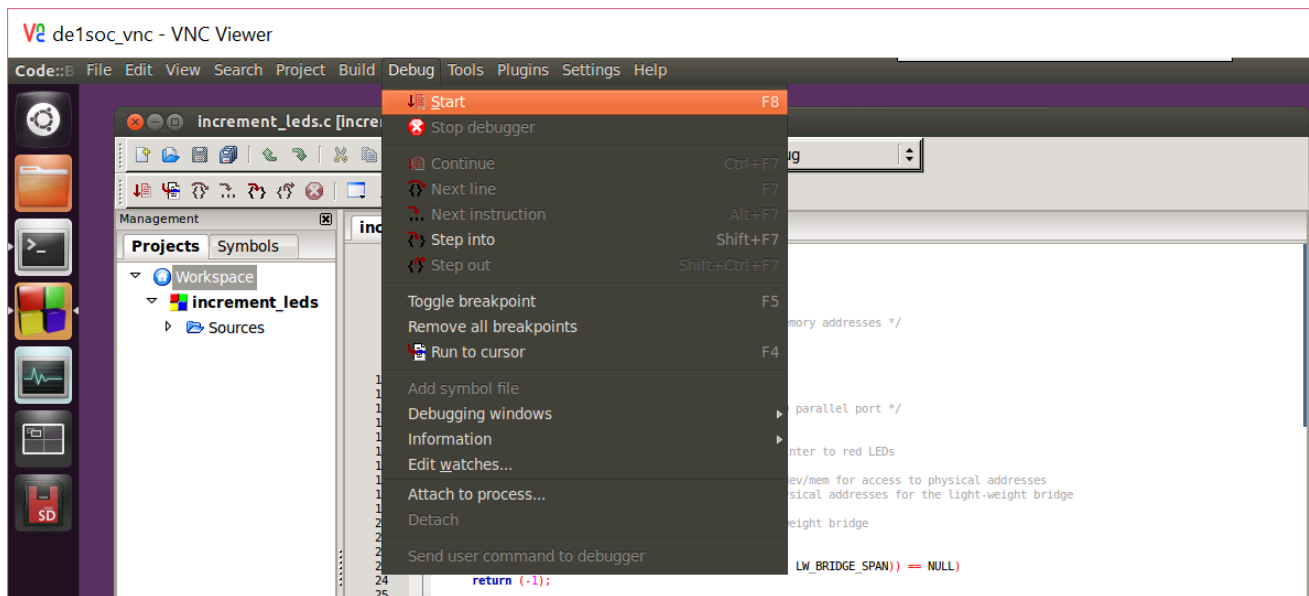


Figure 27. Starting the debugger.

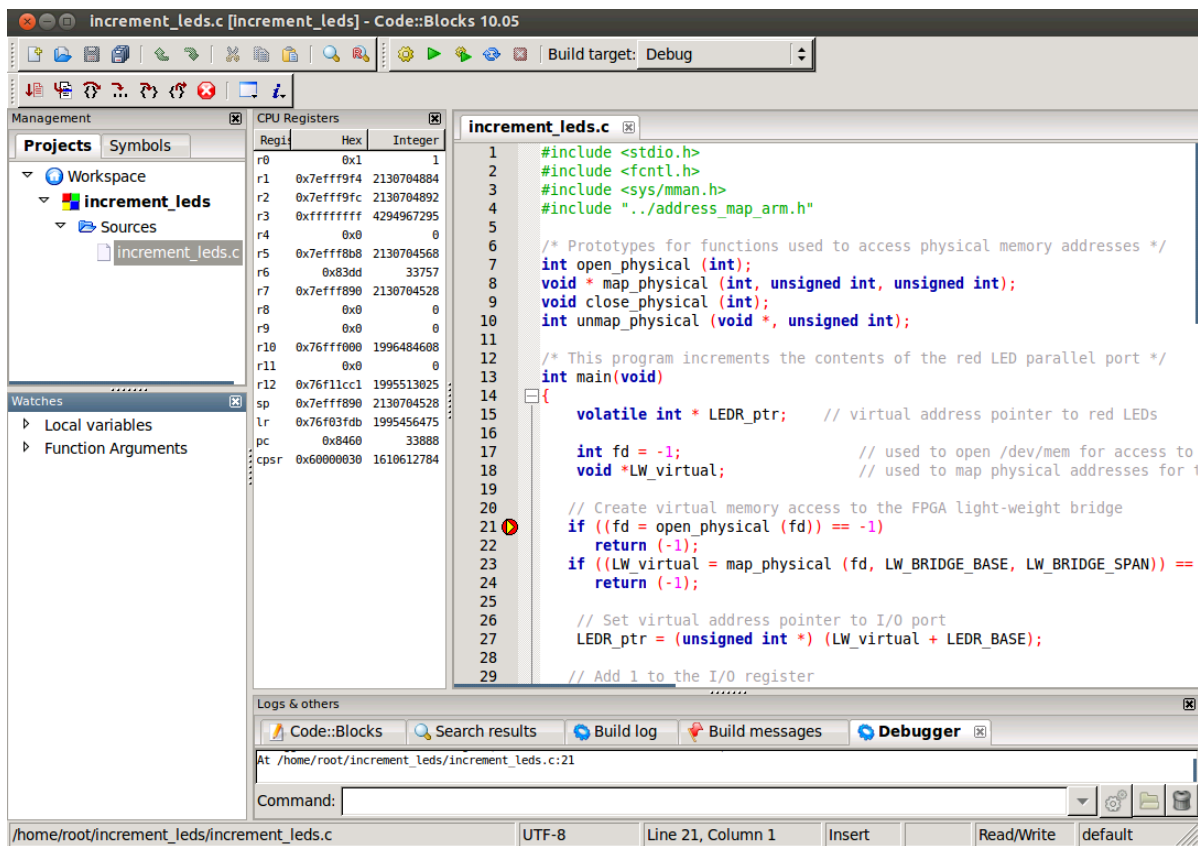


Figure 28. The debugging window.

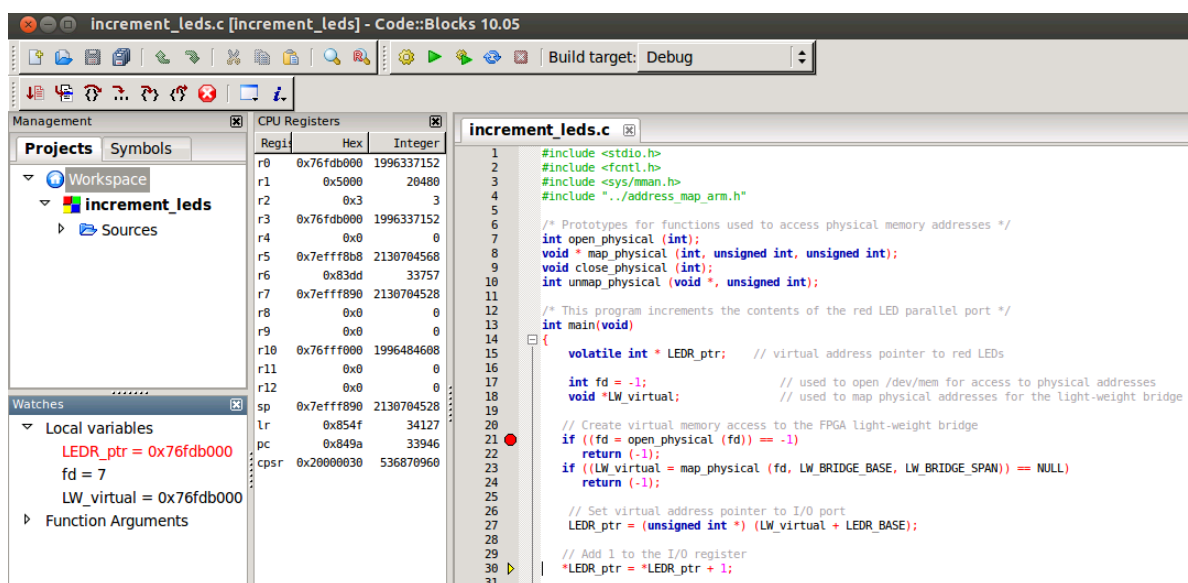


Figure 29. Displaying the values of variables.

Appendix B Include Files

Figure 30 shows the contents of the *include* file *address_map_arm.h* that is discussed in Section 3.3. This 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 TIMER0_BASE        0x00002000
#define TIMER1_BASE        0x00002020
#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 30. The contents of the file *address_map_arm.h*.

Figure 31 shows the contents of the *include* file *interrupt_ID.h* that is discussed in Section 3.4. This file lists the FPGA interrupt line numbers in the DE1-SoC Computer.

```
/* FPGA interrupts */
#define    TIMER0_IRQ          72
#define    KEYS_IRQ           73
#define    TIMER1_IRQ         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 31. The contents of the file *interrupt_ID.h*.

Appendix C Cross Compiling

When a program is compiled for an architecture that is different from that of the system doing the compiling, the process is called *cross-compilation*. In this section we will cross-compile a program for the ARM architecture, to run on the Board's Computer, from a host computer which typically runs on the x86 architecture. To do this, we will use a *gcc* toolchain that comes with the *Altera SoC EDS* suite. Specifically, we will use the `arm-linux-eabi-hf-toolchain`, which can be found in the `/embedded/ds-5/sw/gcc/bin` folder in the Altera SoC EDS installation directory.

We will compile a simple *helloworld* program, the code for which is shown below in Figure 32. Save this code as `helloworld.c` in a folder of your choice on your host computer.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello World!\n");
6
7      return 0;
8  }
```

Figure 32. The helloworld program

As mentioned, we will be using the `arm-linux-gnueabi-hf-toolchain` to compile this program. To start up a shell that includes this toolchain in its path, run the *Embedded Command Shell* batch script, located at `/altera/15.0/embedded/Embedded_Command_Shell.bat`. This will open up the shell, similar to what is shown in Figure 33. Navigate to the folder that contains the `helloworld.c` file by using the `cd` command.

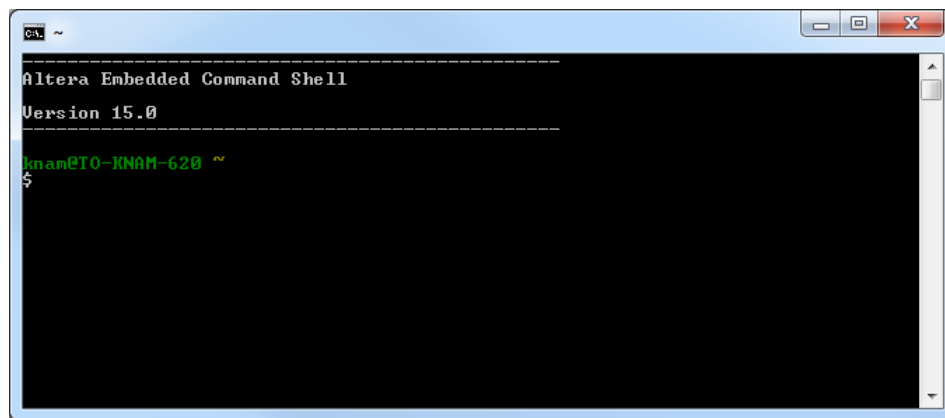
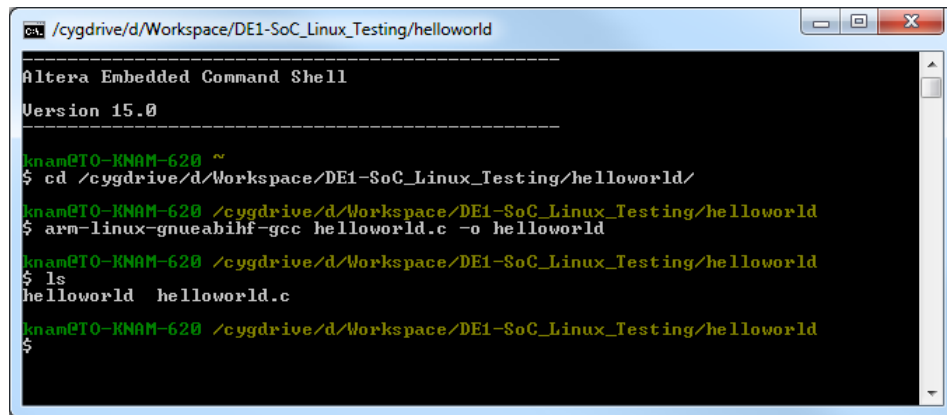


Figure 33. The Embedded Command Shell

Compile the code using the command `arm-linux-gnueabi-hf-gcc helloworld.c -o helloworld`, as shown in Figure 34. This command creates the output file *helloworld*, which is an ARM binary executable that we can copy to our Linux microSD card and execute on the Boards Computer. To copy the executable file you can either use an *ftp* program as discussed in Section 2.7.4, or you can follow the instructions in Section 3.4.3.



```

C:\cygdrive\d\Workspace\DE1-SoC_Linux_Testing\helloworld
-----
Altera Embedded Command Shell
Version 15.0
-----
knam@TO-KNAM-620 ~
$ cd /cygdrive/d/Workspace/DE1-SoC_Linux_Testing/helloworld/
knam@TO-KNAM-620 /cygdrive/d/Workspace/DE1-SoC_Linux_Testing/helloworld
$ arm-linux-gnueabihf-gcc helloworld.c -o helloworld
knam@TO-KNAM-620 /cygdrive/d/Workspace/DE1-SoC_Linux_Testing/helloworld
$ ls
helloworld  helloworld.c
knam@TO-KNAM-620 /cygdrive/d/Workspace/DE1-SoC_Linux_Testing/helloworld
$

```

Figure 34. Cross-compiling the helloworld program

In addition to `arm-linux-gnueabihf-gcc`, the `arm-linux-gnueabihf-` toolchain contains the typical suite of gnu compilation tools such as the C++ compiler (`g++`), linker (`ld`), assembler (`as`), object dump (`objdump`), and object copy (`objcopy`).

Transferring Files to the Linux Filesystem

You may wish to copy over files (such as a program that you want to run on the board) from your host PC to the Linux filesystem. The following sections describe how to do so from Windows and Linux host computers. Note that the host computer must have a microSD card reader.

From a Windows Host PC

When the microSD card is plugged into a Windows host PC, the FAT32 partition of the microSD card is detected. Any files that you move to this partition can be found in the `/media/fat_partition/` directory of the Linux filesystem once Linux boots on the board. Note that this partition will by default contain the files `soc_system.rbf` (an intermediate FPGA programming file used during boot up), `socfpga.dtb` (the device tree file), and `uImage` (the Linux kernel). Under no circumstances should you delete these files, as they are crucial components required to boot Linux.

From a Linux Host PC

When the microSD card is plugged into a Linux host PC, two partitions of the microSD card are detected. The first is the Linux filesystem partition, where you have access to any directory in the Linux directory tree. The second is the FAT32 partition, which gets mounted to `/media/fat_partition/` in the Linux directory tree. You can copy over files from your host PC to either of these partitions. Note that the FAT32 partition will by default contain the files `soc_system.rbf` (an intermediate FPGA programming file used during boot up), `socfpga.dtb` (the device tree file), and `uImage` (the Linux kernel). Under no circumstances should you delete these files, as they are crucial components required to boot Linux.

Appendix D FPGA Configuration

A special mechanism built into the Cyclone V SoC allows software running on the ARM processor (such as the Linux OS) to program the FPGA. The UP Linux distributions contain drivers for this mechanism, allowing us to program the FPGA from the CLI. The following sections describe how to use this mechanism.

Creating an RBF Programming File

The FPGA programming mechanism accepts an input FPGA bitstream in the *Raw Binary File* (.rbf) file format. This means that once you compile your circuit using Quartus, which outputs the FPGA bitstream in the *SRAM Object File* (.sof) file format, you must convert the .sof file into a .rbf file. This is done using Quartus's *Convert Programming File* tool, and the steps are described below.

1. Launch the Convert Programming File tool by selecting `File > Convert Programming Files....`
2. Select `Raw Binary File (.rbf)` as the Programming file type.
3. Select `Passive Parallel x16` as the Mode.
4. Specify the destination file name in the `File name` field.
5. Click and highlight `SOF Data` then add the .sof file that you wish to convert by clicking `Add File....`
6. Click and highlight the newly added .sof file in the list, then select `Properties`. You should see the window shown in Figure 36. Enable file compression by ticking the checkbox as shown, then press OK.
7. We are now ready to generate the .rbf file. Click `Generate`. If all goes well, you will see the success message shown in Figure 37.
8. Finally, we can transfer the .rbf file to the Linux file system using the instructions provided in Section 3.4.3.

Programming the FPGA

The Linux distributions expose the FPGA as the device file located at `/dev/fpga0`. In addition, Linux provides files in `/sys/class/fpga/` and `/sys/class/fpga-bridges/` for probing the status and configuring settings of FPGA-related components. We will make use of these file-based interfaces to program the FPGA, using the steps described below.

1. Ensure that the MSEL switches on the board have been configured to `MSEL[4:0] = 5'b01010`.
2. Before we reprogram the FPGA with new components, we must first disable the FPGA-HPS bridges to avoid unpredictable behavior. Disable them using the following commands:
 - `echo 0 > /sys/class/fpga-bridge/fpga2hps/enable`
 - `echo 0 > /sys/class/fpga-bridge/hps2fpga/enable`
 - `echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable`

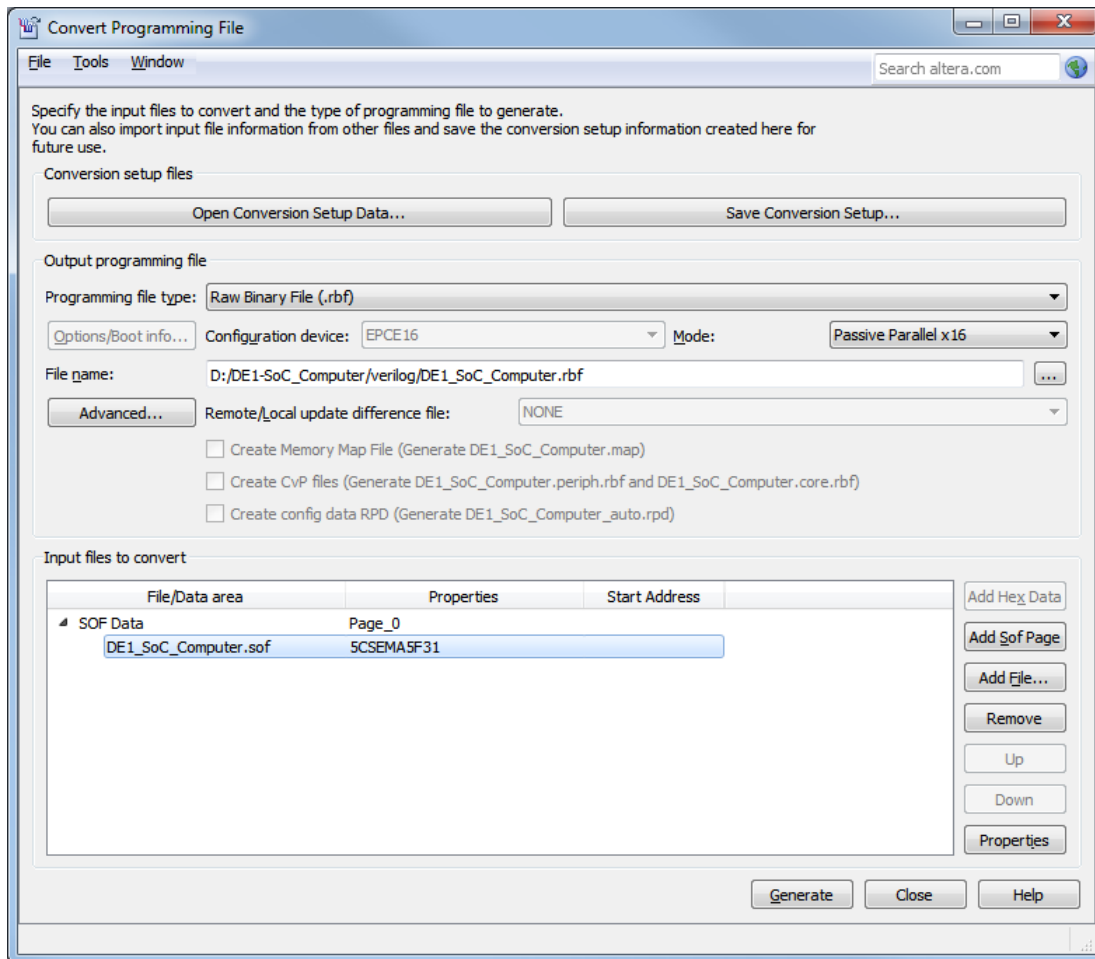


Figure 35. The Convert Programming File Tool.

3. Load the .rbf into the FPGA device using the command:

- `dd if=<filename> of=/dev/fpga0 bs=1M`

where <filename> is the full path to your .rbf file.

4. Re-enable the required FPGA-HPS bridges using the following commands:

- `echo 1 > /sys/class/fpga-bridge/fpga2hps/enable`
- `echo 1 > /sys/class/fpga-bridge/hps2fpga/enable`
- `echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable`

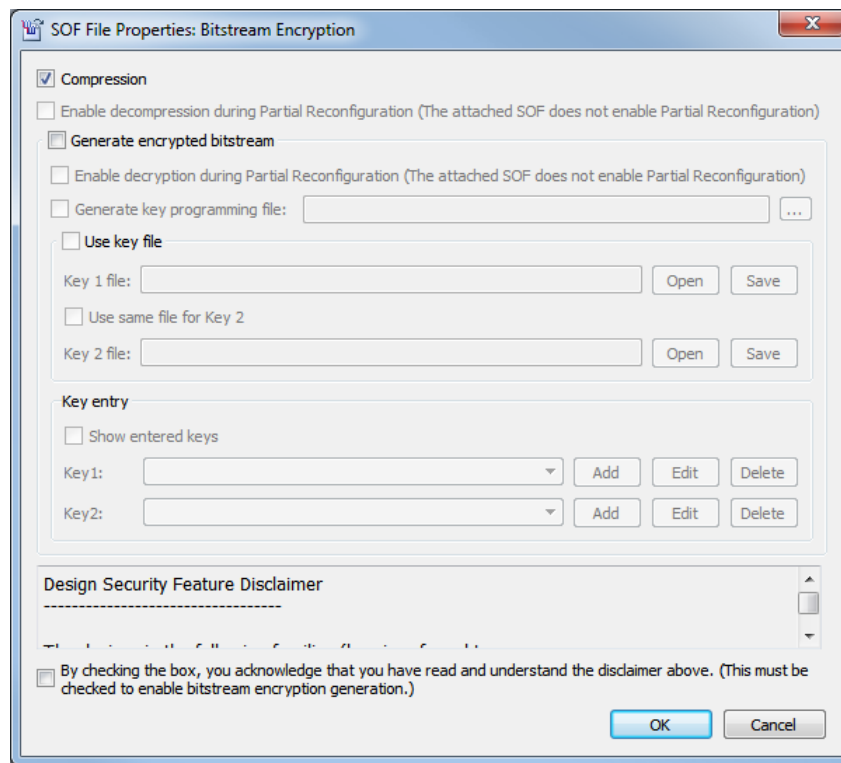


Figure 36. Enabling file compression.

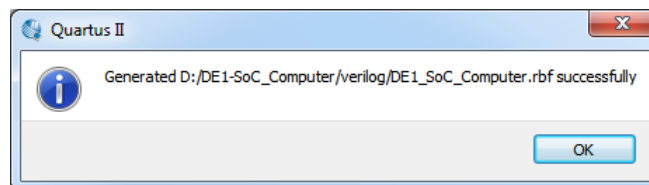


Figure 37. The .rbf file successfully generated.

Changing the Default FPGA Programming File

While the Linux distribution boots, scripts are executed to initialize various Linux components. One of these scripts, located at `/etc/init.d/programfpga`, programs the FPGA with the default programming (.rbf) file. If you open the script using a text editor such as `vi`, you will see that the script executes the FPGA programming commands described in Section 3.4.3. To change the default FPGA programming file, edit the line `dd if=<.rbf file> of=/dev/fpga0 bs=1M` to specify an .rbf file of your choosing.

Copyright © Intel Corporation.