

1 Introduction

This tutorial describes a release of Linux* which is available for a variety of embedded systems that feature an Intel® Cyclone® V SoC device. This Linux distribution can be used on the following development and education (DE-series) boards: DE1-SoC, DE10-Standard, and DE10-Nano. For this tutorial we will assume that the reader is using the DE1-SoC board. If you are using a different board, then some minor adjustments to the instructions given in the tutorial may be needed. We will make note of such cases in various sections of the tutorial.

Linux runs on the ARM* processor that is part of the Cyclone V SoC device. In this tutorial we show how Linux can be stored onto a microSD* memory that can be inserted into the DE-series board and booted by the ARM processor. We also show how software programs can be developed that run on the ARM processor under Linux, and make use of the hardware resources on the DE-series board. These resources include peripherals in the hard processor system (HPS), and custom hardware peripherals implemented within the FPGA in the 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 mentioned above. 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 would typically be either a desktop computer or laptop. The host computer provides a user-interface for connecting to your DE-series board. You will use the host computer to send commands to the Linux running on the DE-series board, and to see the results produced by those commands.
- Mini-USB cable, Ethernet cable, and/or WiFi USB adaptor for connecting the DE-series board to the host computer
- MicroSD card (8 GB or larger)

Optional:

- Intel FPGA SoC Embedded Design Suite (required for Appendix G).
- Intel Quartus® Prime Software (required for Appendix H).

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 Intel Cyclone® V SoC Devices

The DE1-SoC, DE10-Standard, and DE10-Nano boards feature an Intel Cyclone V SoC device, which contains two main components: a *Hard Processor System* (HPS), and an *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 Linux* Distribution Image

A number of Linux distributions are available for the 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). Linux distributions are provided in the *.img* (image) file format, which can be written onto a microSD card and booted on the DE-series board. For this tutorial we will assume that the reader is using the *DE1-SoC-UP* Linux distribution for the DE1-SoC board. The corresponding image file *DE1-SoC-UP-Linux.img* can be downloaded from the Intel FPGA University Program website (look for the latest version under `Materials > Software > Embedded Linux`). If you are using a different board, then you will need to use the specific version of the Linux image that is provided for that board.

The *DE1-SoC-UP* Linux distribution 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 programs. We will make extensive use of this toolchain to compile programs in Section 3. Another feature is the automatic programming of the Cyclone V SoC device that takes place during the process of booting the *DE1-SoC-UP* operating system (OS). The OS configures the SoC device into a circuit called the *DE1-SoC Computer* system. This computer system contains a dual-core ARM Cortex A9 processor, as well as IP cores that communicate with the peripheral devices found on the DE1-SoC board, such as the switches, LEDs, pushbuttons, VGA, audio, Ethernet, and USB. In Section 3.3, we will show how to write programs that communicate with Parallel I/O 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*, which is available from the Intel FPGA University Program website.

2.3 Preparing the Linux* microSD* Card

The DE1-SoC board is 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 *DE1-SoC-UP-Linux.img* 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.

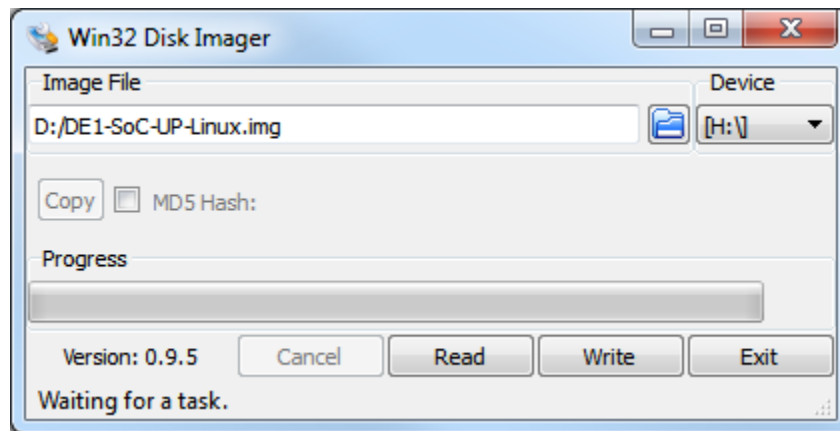


Figure 1. The Win32 Disk Imager program.

2. Select the drive letter corresponding to the microSD card under *Device*, as indicated in Figure 1.
3. Select the *DE1-SoC-UP-Linux.img* image under *Image File*, as shown in Figure 1. This image file can be found on the Intel FPGA University Program webpage alongside this tutorial.
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 Setting up your DE-series Board for use with Linux

First ensure that your DE-series 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 underside of 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.

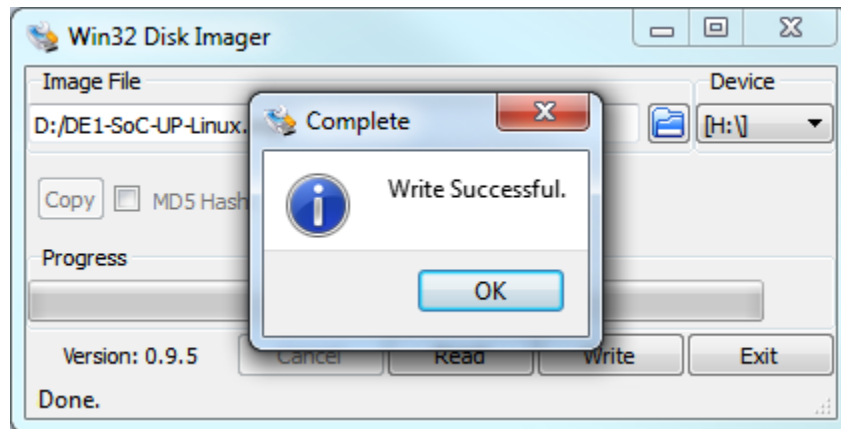


Figure 2. Writing a file to the microSD card using Win32 Disk Imager.

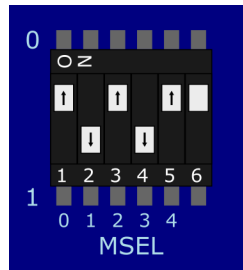


Figure 3. Configuring the MSEL switches of your DE-series board.

2.5 Connecting your DE-series Board to a Host Computer

Before booting Linux, you should first connect your DE-series board to a host computer. There are two main methods of communicating between the DE-series board and the host computer: using a *USB cable*, and using a *network* to connection. Each method is described below.

2.6 Connecting to the Host Computer using a USB Cable

The Linux image has been configured to send and receive text to/from a host computer via a USB cable. The USB cable is connected to a *UART-to-USB* port, which can be used to send/receive text characters. On the host computer, we can use a *terminal* program to display this text. Various *terminal* programs are available; we will be using *putty*, which is available for free download on the Internet for both Windows and Linux host computers.

In the following discussion we assume that you have installed Putty on your host computer. If you use a different terminal program, then the instructions below would need to be modified accordingly. Connect the UART-to-USB port of your DE-series board to your host computer using the mini-USB cable supplied with the board. To identify the correct port, look for a small label on the board that reads UART. It can be found in the area of the board near

the RJ45 Ethernet port. If this is your first time connecting to the UART-to-USB port, then 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 Microsoft* Windows* Host Computer

You should read this section if your host computer is running a Microsoft Windows operating system, such as *Windows 10*. On a Windows host computer, serial communication ports 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 the Windows *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.

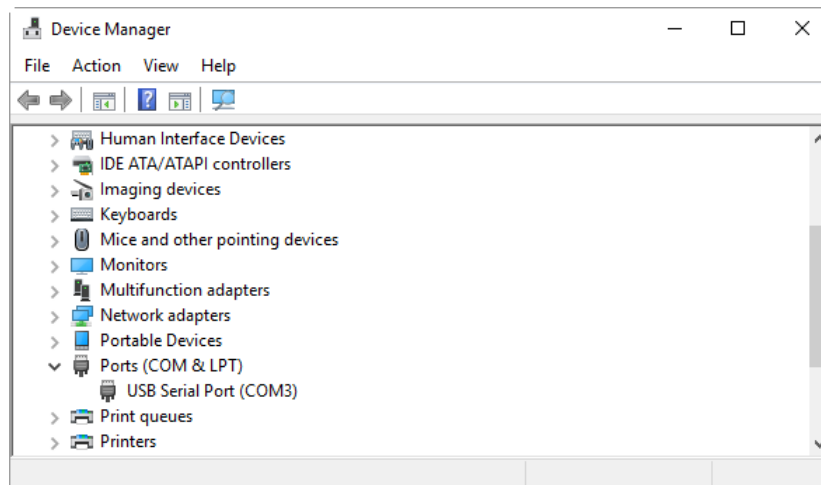


Figure 4. Determining the COM port of the UART-to-USB connection in Device Manager.

2.6.2 Using a Linux* Host Computer

You should read this section if your host computer is running a version of Linux. 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`.



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.

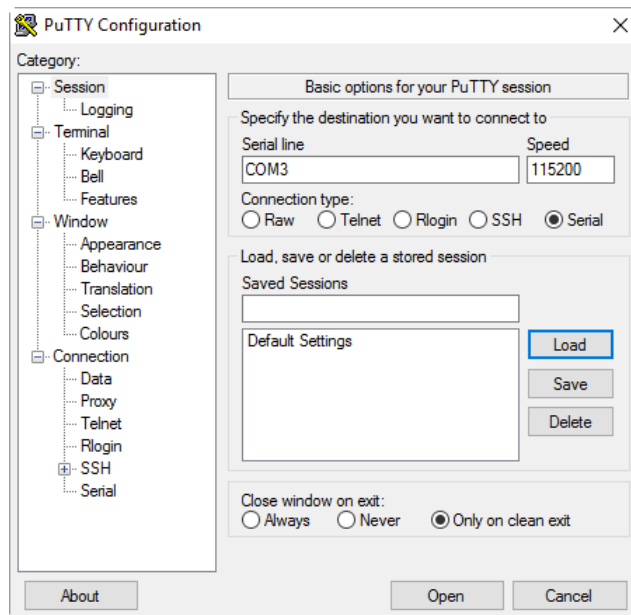


Figure 6. Putty’s main window.

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 your DE-series 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 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.

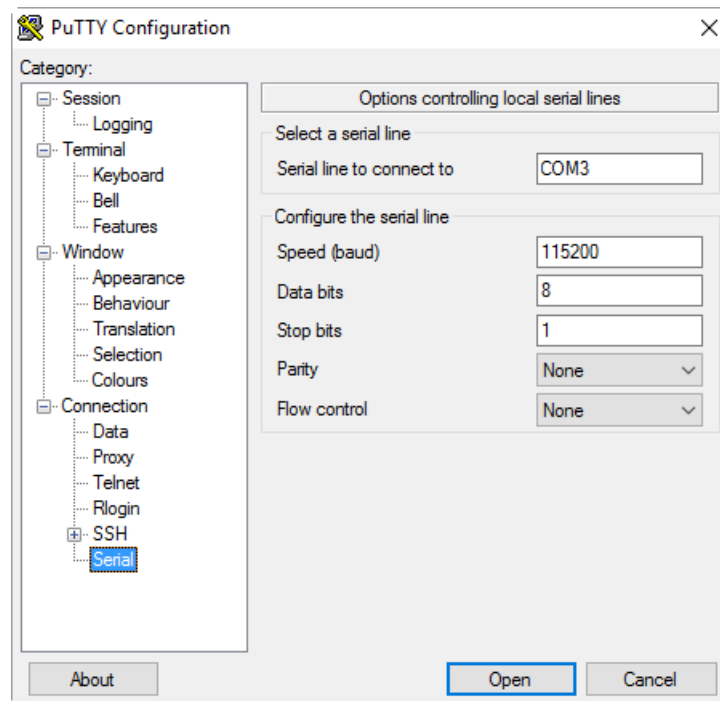


Figure 7. PuTTY's configuration window for serial communication settings.

2.7 Connecting to the Host Computer using a Network

The Linux image has 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 DE-series 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 DE-series board: using an Ethernet cable, and using a WiFi adapter. Both methods are discussed below.

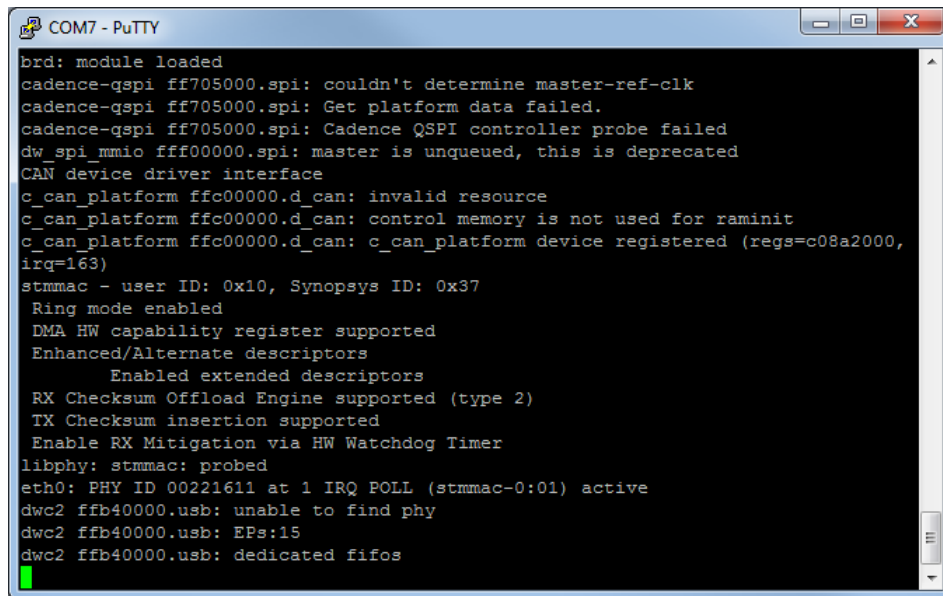


Figure 8. Putty terminal displaying text output as the Linux kernel boots.

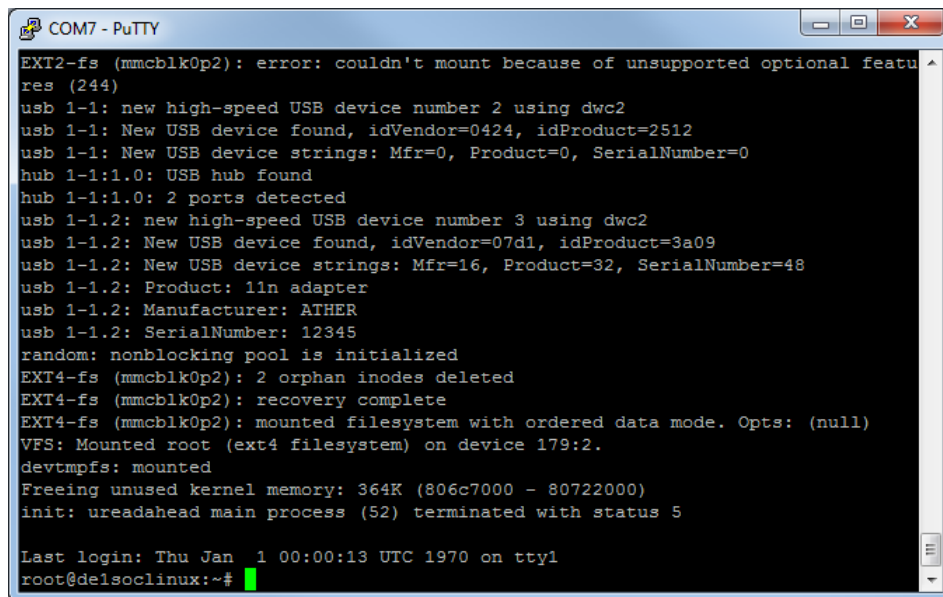


Figure 9. The Linux command line prompt showing the root ('#') logon.

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 DE-series 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 DE-series board is 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 DE-series 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 DE-series board. In most cases the best option is to change the IP address of the DE-series board so that it uses the same subnet as the host computer. But it is also possible to change the IP address of your host computer is that is preferred. The procedures for changing IP addresses are described below.

Changing the IP address of a DE-series Board

To change the IP address of your DE-series 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 DE-series board execute the `ifconfig` command. The `ifconfig` command shows you the current IP address(es) that are assigned to the board. You can use `ifconfig` to change the IP address. For example, let's assume that your host computer's IP address is 192.168.86.33. Then, you could use the command `ifconfig eth0 192.168.86.123` to change your board's IP address. In this command 123 is just an example—it can be any number that is not already being used in your router's subnet. Note that `eth0` is the name of an ethernet port that is available to Linux.

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 your Host Computer

Changing the IP address of your host computer is not usually the preferred approach, but we describe it here for completeness. 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.

2.7.2 Connecting to the Host Computer using a WiFi Adapter

To make a network connection to your DE-series 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 DE-series board to connect to the desired WiFi network.

Linux supports 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 your DE-series 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. Your DE-series 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 your DE-series 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 DE-series 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 DE-series 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 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.

The Linux image includes 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.

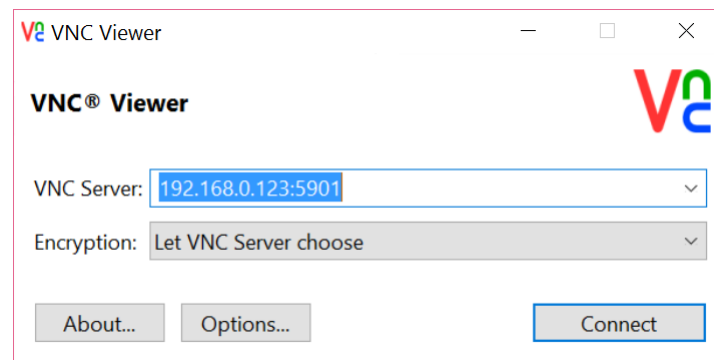


Figure 10. The RealVNC opening dialog.

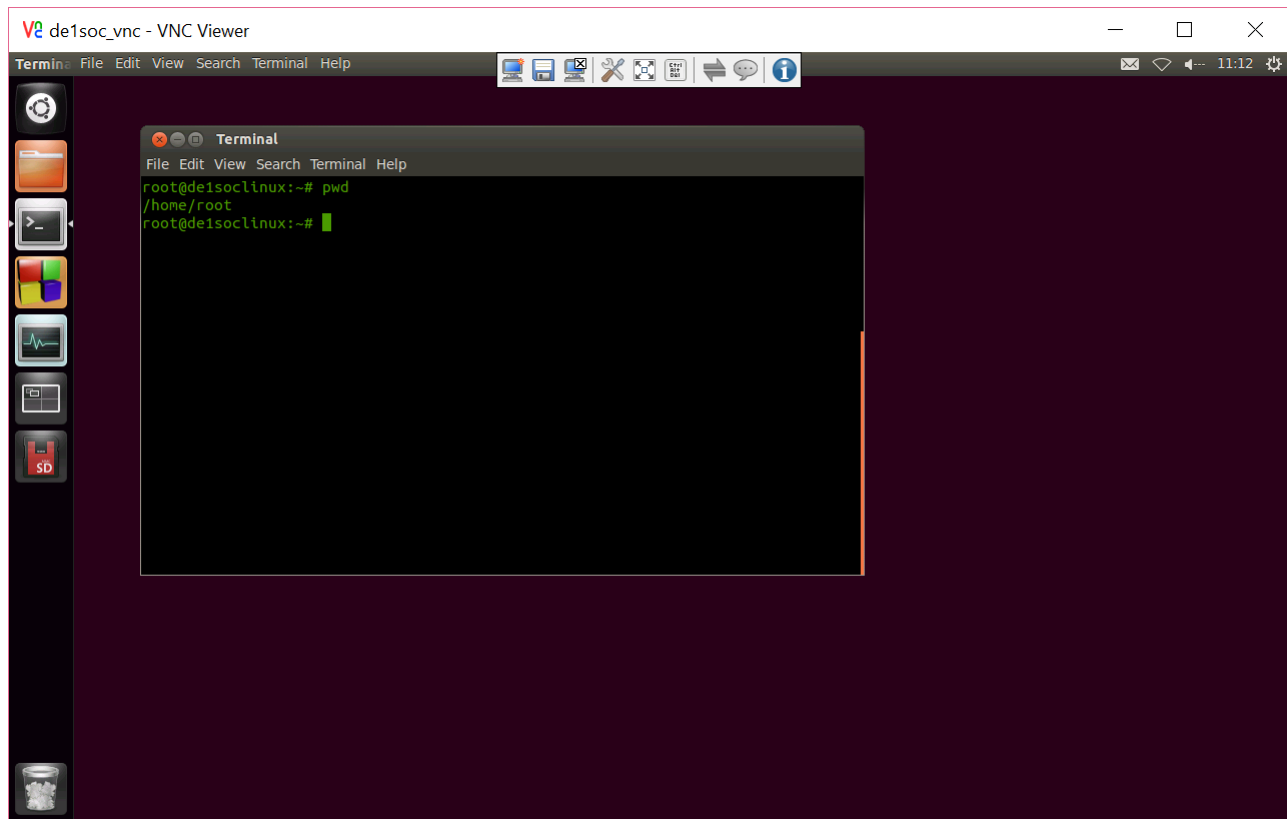


Figure 11. The main RealVNC window.

2.7.4 Transferring Files to/from the Host Computer

The Linux image includes an *FTP* Server that can be used to transfer files between the host computer and the DE-series board. The FTP server uses the secure FTP (SFTP) protocol and uses *Port 22* for 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 your DE-series board to a network, it is possible to provide Internet access to Linux applications. For example, the Linux image includes 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 your DE-series Board

In this section you will learn how to develop programs that can run under Linux on your DE-series board. There are two options for developing a Linux program in a language such as C for your DE-series board. The first is to compile the C code by using the `gcc` compiler that is provided with the Linux running on the board. This approach is called *native compilation*. This is the method that we use in this tutorial. The second option is to 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 G.

To perform native compilation you need to compile your C code by using the `gcc` compiler provided with Linux that is running on the board. However, this process does not require you to actually *type* the C code with a text editor running under Linux. Instead, you could type your C source code on a host computer, such as a Windows PC, and then transfer the file onto your Linux filesystem (microSD card) to compile it. This approach is illustrated in Figure 12. It shows a text editor running on the host computer, which is being used to type the contents of a C source-code file. In this case the *gvim* editor is being used, but you could run whatever text editor you prefer on the host computer. Figure 12 also depicts an *ftp* program being used to transfer the source-code file to Linux. In this *ftp* program, files on the host computer are listed in the left-hand side of the window, and files on the Linux filesystem (microSD card) are listed on the right-hand side. In the figure we are running the *Filezilla** program, but any *ftp* client on the host computer can be used. Finally, the figure depicts a *Terminal* window connected to Linux. The Terminal provides access to the Linux *command line interface* (CLI). The CLI allows you to execute Linux programs, such as the C-compiler, on the DE-series board. The Terminal window in this case is provided by *putty*. It communicates with the DE-series board via a USB cable, discussed in section 2.6. The *ftp* program communicates with the board via a Network connection, discussed in section 2.7

An alternative to the setup in Figure 12 is to use the VNC Viewer as depicted in Figure 11. In this case you would not need to use the Terminal window that is connected via USB, because an equivalent Terminal window can be opened directly in the VNC Viewer window. You could choose to run one of the editors provided with the Linux image within the VNC Viewer, to create source-code files directly on the Linux filesystem (microSD card). Or, you could still create files on the host computer and then transfer them to Linux via *ftp*.

3.1 Native Compilation on the DE1-SoC Board

As mentioned previously, 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 tool-chain.

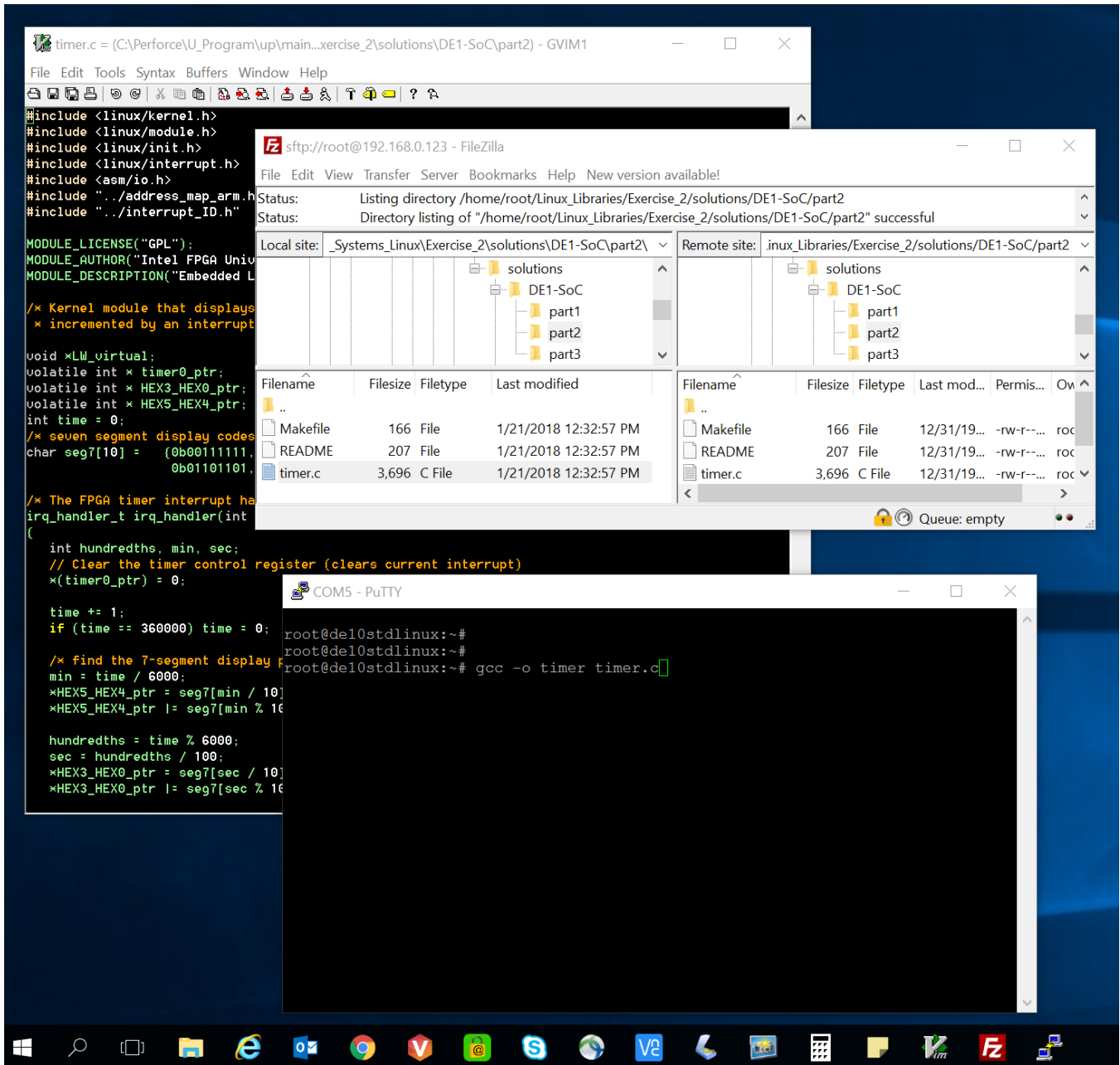


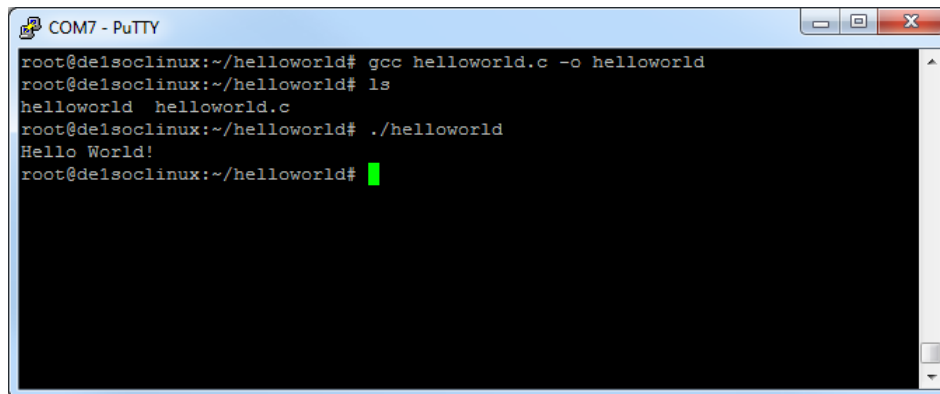
Figure 12. Using a host computer to write code, and ftp to transfer to Linux.

To demonstrate native compilation, we will compile a simple "hello world" program. The code for this program is shown in Figure 13. You can also find the code in `/home/root/tutorial_files/helloworld/helloworld.c` of the Linux filesystem.

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

Figure 13. The helloworld program

You can compile code using the Linux command-line interface. If you are using a USB cable to connect to the DE1-SoC 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/tutorial_files/helloworld`. Compile the program using the command `gcc helloworld.c -o helloworld`, as shown in Figure 14. 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!", and then exits, as shown in Figure 14.



```
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 14. 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 *DE1-SoC-UP-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 *Using the DE1-SoC Computer with 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 (*lwahps2fpga*) 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 *DE1-SoC-UP* 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 + offset`. The offset of the red LED port is 0, so its address is `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 15.

The code for the application program is given in Figure 16. 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/tutorial_files/increment_leds/increment_leds.c`. You can compile the code using a command such as `gcc -Wall increment_leds.c -o increment_leds`, and then run the program with the command `./increment_leds`. The code is described below.

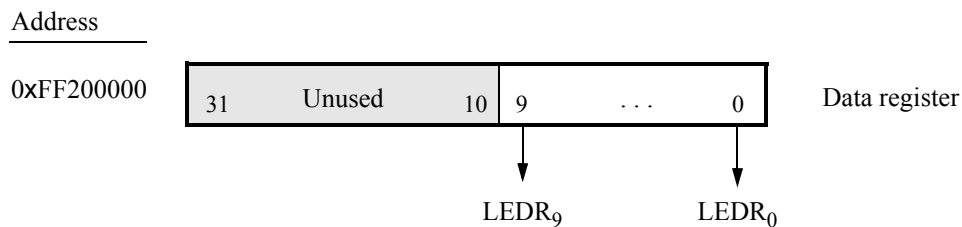


Figure 15. The LED parallel port.


```

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)))
23         return (-1);
24
25     // Set virtual address pointer to I/O port
26     LEDR_ptr = (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 16. 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 17. 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 *lwhps2fpga*, 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

```

/* 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;
}

```

Figure 17. Functions for managing physical memory addresses (Part a).

```

/* 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 17. 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 18. 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 18. 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 19. 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 19, 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     *LEDR_ptr = 0; // Turn off LEDs and de-register irq handler
40     iounmap (LW_virtual);
41     free_irq (KEYS_IRQ, (void*) irq_handler);
42 }
43 module_init(initialize_pushbutton_handler);
44 module_exit(cleanup_pushbutton_handler);

```

Figure 19. 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/tutorial_files/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 20. 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 `<module_name>.ko` file is a special type of executable program used only for kernel modules.

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

Figure 20. Kernel module makefile

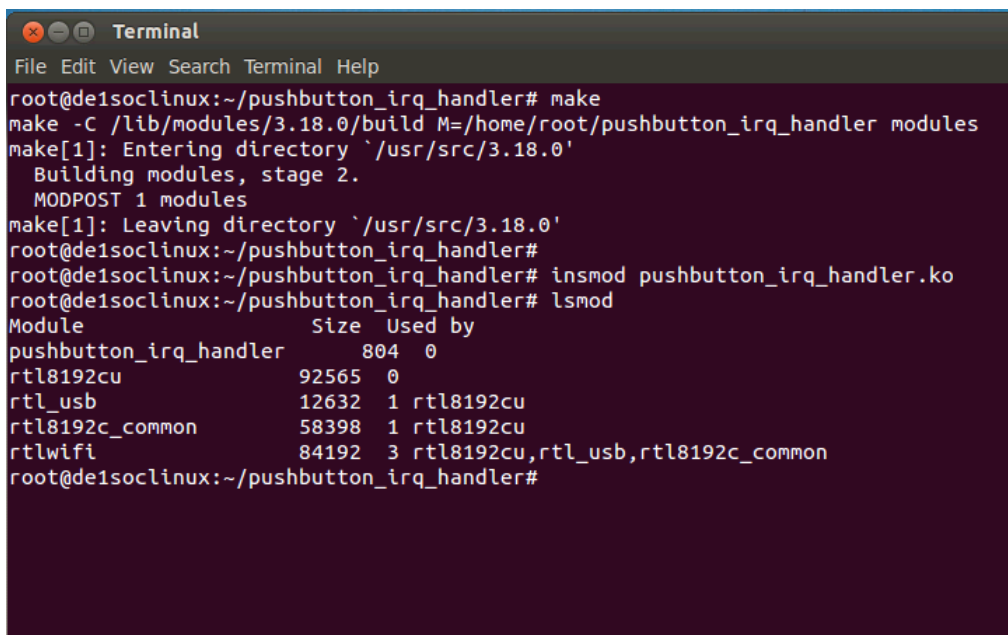
3.4.3 Using the `make` Tool

The `make` tool provides a convenient mechanism for compiling and building programs, because `make` only compiles the parts of a program that have been updated since the last time the program was compiled. This mechanism relies on the ability of `make` to check the time- and date-stamp of files and compare them to the current time and date of the Linux system. On your DE-series board this mechanism can fail to work properly, because the board does not have a battery that would allow it to save the current date and time when powered off. If the board is reset by pressing the power button, then the Linux system will reboot and the date and time will be reverted to to the "Linux epoch", which is January 1, 1970 00:00:00. After the date and time have been reverted the `make` program will not be able to properly check time stamps on files when you compile programs—your files will appear to have dates/times that are “in the future”. One way to address this issue is to manually update the date and time after Linux has been rebooted, to the current date and time. For example, if the date and time is October 15, 2017 at 2:10 pm, you would type the Linux command `date -s "10/15/2017 14:10:00"`. If you consistently set the date and time whenever Linux is rebooted, the `make` program should work properly.

3.4.4 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 21. 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.



```

Terminal
File Edit View Search Terminal Help
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 21. Inserting and removing the kernel module

3.4.5 Using Intel® FPGA Device Drivers

In addition to being able to create your own kernel modules, as discussed above, the *DE1-SoC-UP* Linux distribution provides a number of pre-built kernel modules that are available for communicating with hardware modules in the DE1-SoC Computer. These pre-built modules are listed in Table 1.

Each of the kernel modules listed in Table 1 provides a *character device driver* for accessing a port in the DE1-SoC Computer. This means that each module has a file-based interface that can be used to read information from the driver, or write information to it. The file-based interface is provided in the folder `/dev/IntelFPGAUP`. One way to read from a driver is to use the Linux `cat` command. For example, to read the states of the pushbutton switches you can type `cat /dev/IntelFPGAUP/KEY`. Usage information for each driver can be found by writing `--`

Kernel module	Description
KEY	Used to access the pushbutton KEY port
SW	Used to access the slide switch SW port
LEDR	Used to access the red light LEDR port
HEX	Used to access the seven-segment HEX display port
video	Used to access the VGA video-out port
audio	Used to access the digital audio port
accel	Used to access the 3-D accelerometer port

Table 1. Pre-built kernel modules

to the driver. For example, to see how to use the video driver you could type the Linux command `echo -- > /dev/IntelFPGAUP/video`.

The drivers listed in Table 1 are not inserted into the Linux kernel during the boot process. To insert a driver, navigate to the directory `/home/root/Linux_Libraries/drivers`. To insert a specific driver, you can use the Linux command `insmod`. For example to insert the KEY driver you would type `insmod KEY.ko`. To insert all available drivers, you can execute the script `load_drivers`. Similarly, you can then remove an individual driver by using the command `rmmmod`, or remove all of the drivers by executing the script `remove_drivers`.

For convenience, a set of *wrapper* functions is provided in the C language for use with each character device driver. To use these functions in a program, you need to provide the statement `#include "IntelFPGAUP/xxx.h"` in your C code, where *xxx* is the name of the driver from Table 1 (KEY, SW, ...). In addition, you have to append the option `-lintelfpgaup` to the `gcc` command when compiling your code. The contents of the files *xxx.h*, which list all of the available wrapper functions, are shown in Appendix C. An example of a C program that uses these wrapper functions is given in Appendix D. If this program is stored in a file name *draw_lines.c*, then you would compile it with a command such as `gcc -Wall -o draw_lines draw_lines.c -lintelfpgaup`. The wrapper source code files and examples can be found in the directory `~/Linux_Libraries/C4DE`.

A set of wrapper functions is also provided in the Python* language for use with each character device driver. To use these functions in a Python program you need to include the statement `import xxx`, where *xxx* is the name of the device from Table 1. You also need to have the proper setting in your `PYTHONPATH` environment variable so that the Python interpreter can locate the wrapper functions. The contents of the Python wrappers *xxx.py*, which list all of the available wrapper functions, are shown in Appendix E. An example of a program that uses these wrapper functions is given in Appendix F. The wrapper source code files and example programs are included in the directory `~/Linux_Libraries/PyDE`.

To check your `PYTHONPATH` variable you can execute the Linux command `echo $PYTHONPATH`. If the path variable does not include the directory `~/Linux_Libraries/PyDE/src`, then in the file `~/ .bash_profile` add the line `export PYTHONPATH=:/home/root/Linux_Libraries/PyDE/src`. This statement assumes that your home directory is `/home/root`. If not, then substitute the appropriate directory name. Finally, include the new setting in your current login session by executing the command `source ~/ .bash_profile`.

Drivers for the DE10-Standard Board

If you are using the DE10-Standard board, then all of the device drivers listed in Table 1 are available, as well as their corresponding wrappers. In addition, there is a device driver called `LCD`, which controls the 128 × 64 LCD display on the DE10-Standard board. The wrappers provided for the `LCD` driver are identical to those of the video driver in Table 1.

Drivers for the DE10-Nano Board

If you are using the DE10-Nano board, then all of the device drivers listed in Table 1 are available, except for the `HEX` driver. Also, for this board the `LEDR` driver, and corresponding wrappers, is named `LED`.

Appendix A Using Code ::Blocks

If you have a network connection to your DE-series board, 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*. 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 22. 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 23, 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.

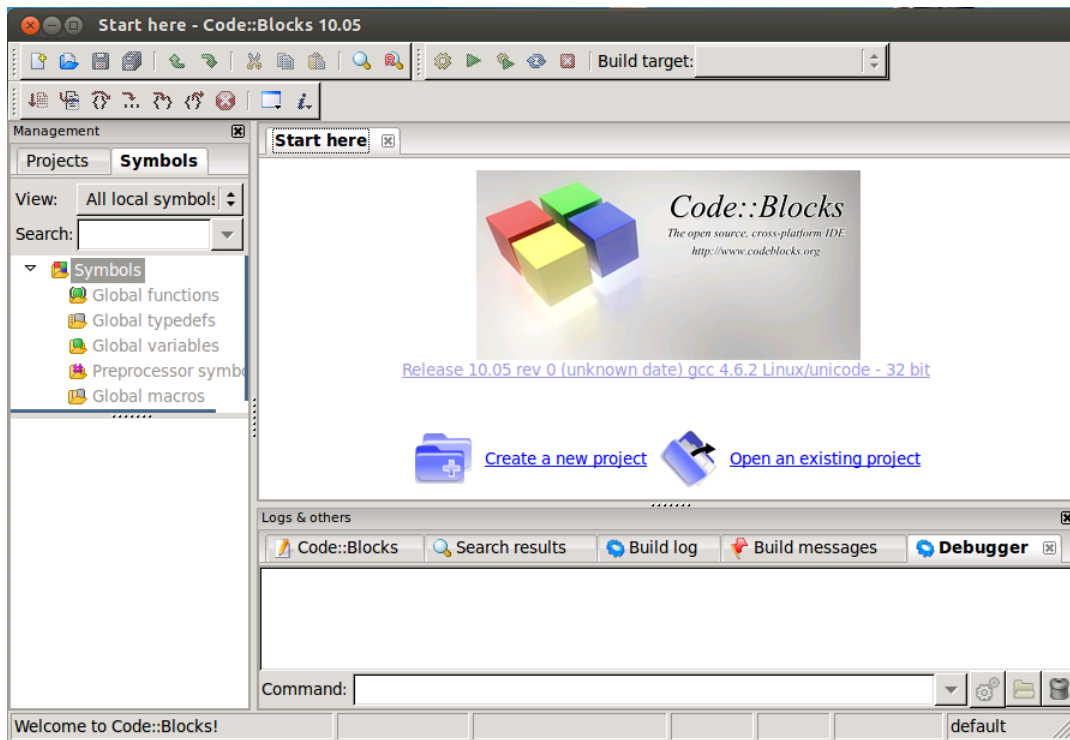


Figure 22. The main Code ::Blocks window.

As indicated in Figure 24 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 25, and then click the *Open* button. In the dialog that opens, illustrated in Figure 26, 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.

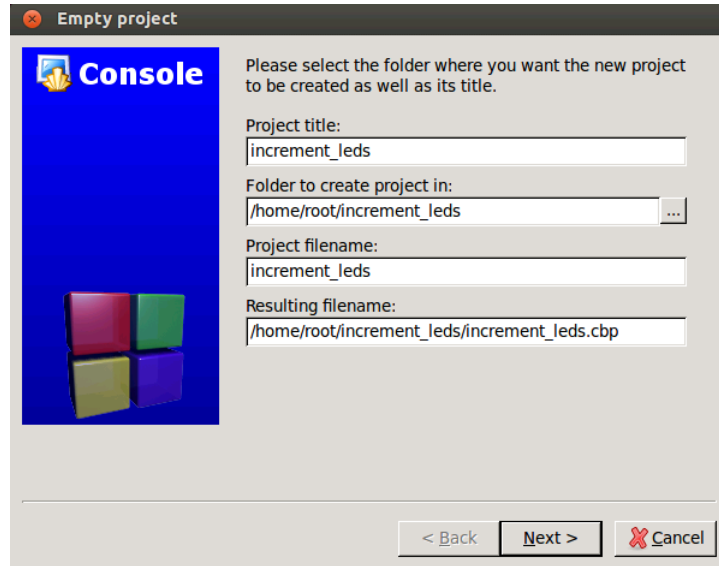


Figure 23. The Empty project dialog.

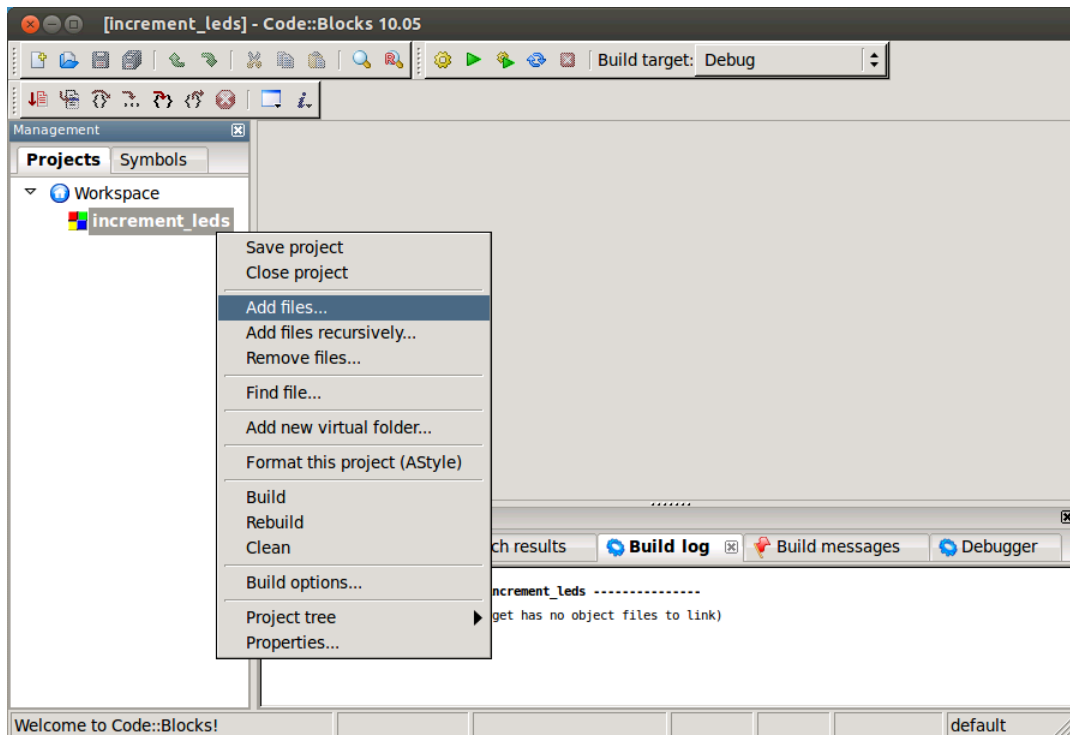


Figure 24. Adding source-code files to the project.

Click to the right of the line of code that calls the function `open_physical`, as shown in Figure 27, 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 28. (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 29 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 30. 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.

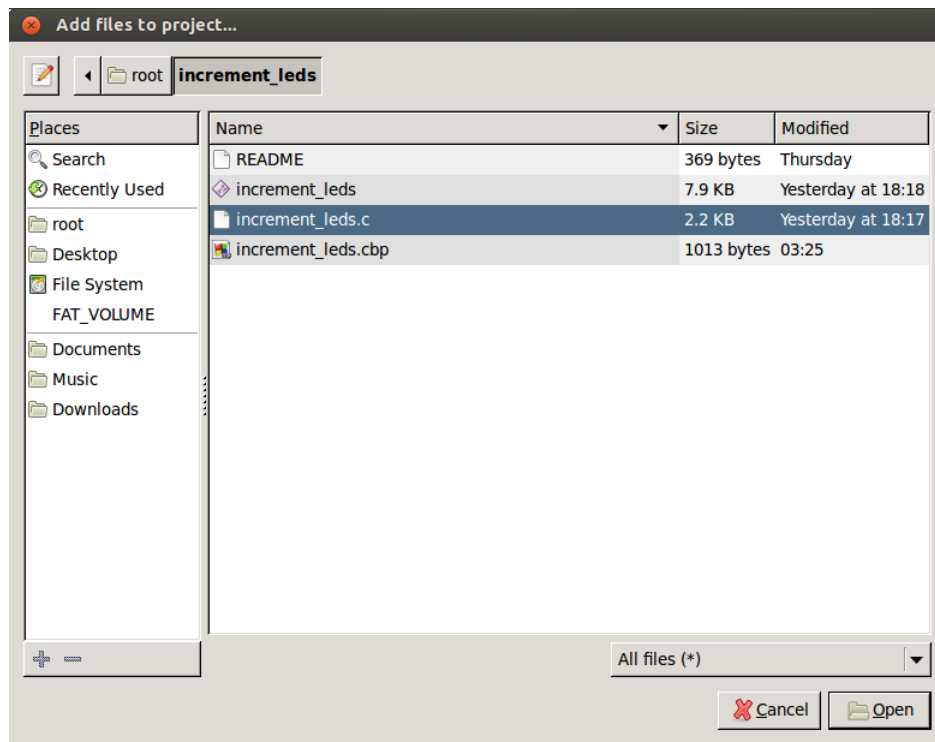


Figure 25. Selecting the *increment_leds.c* file.

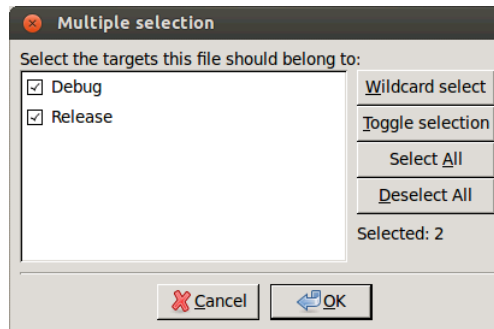


Figure 26. Selecting build targets.

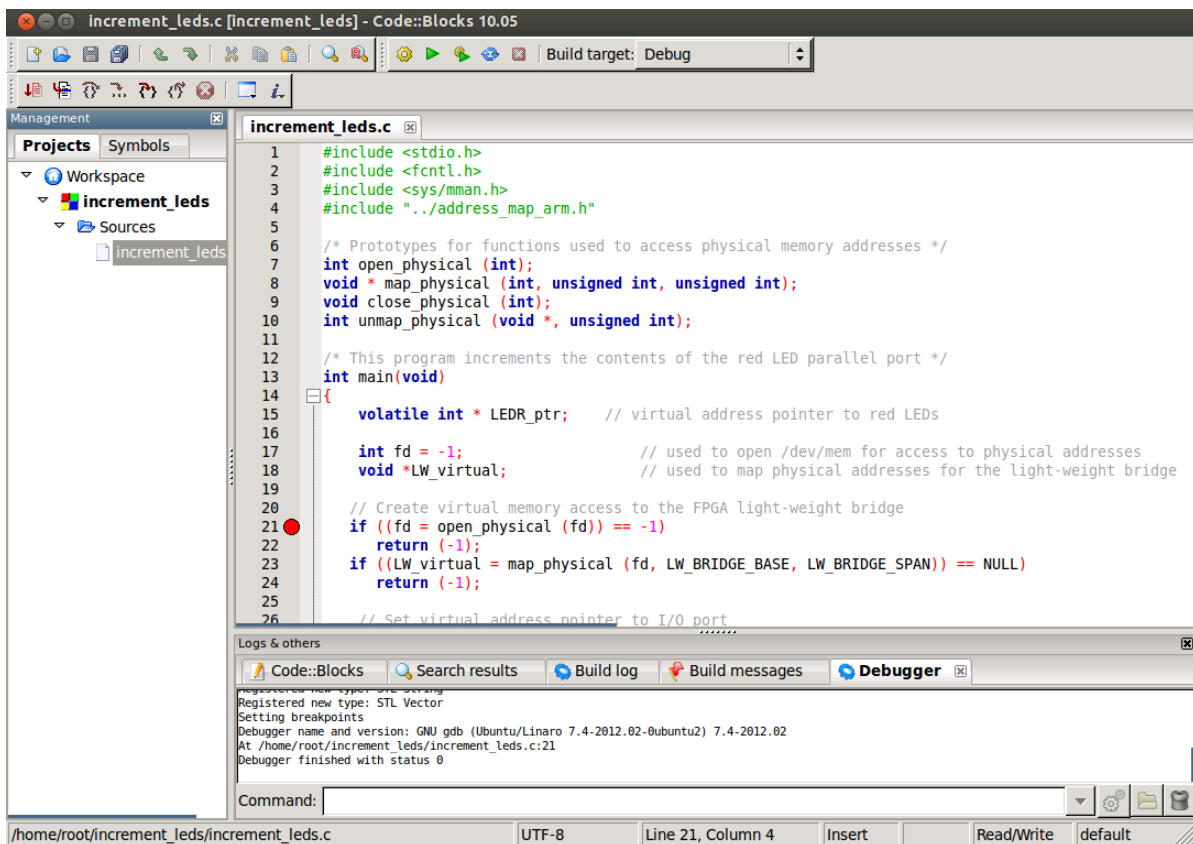


Figure 27. Setting a breakpoint.

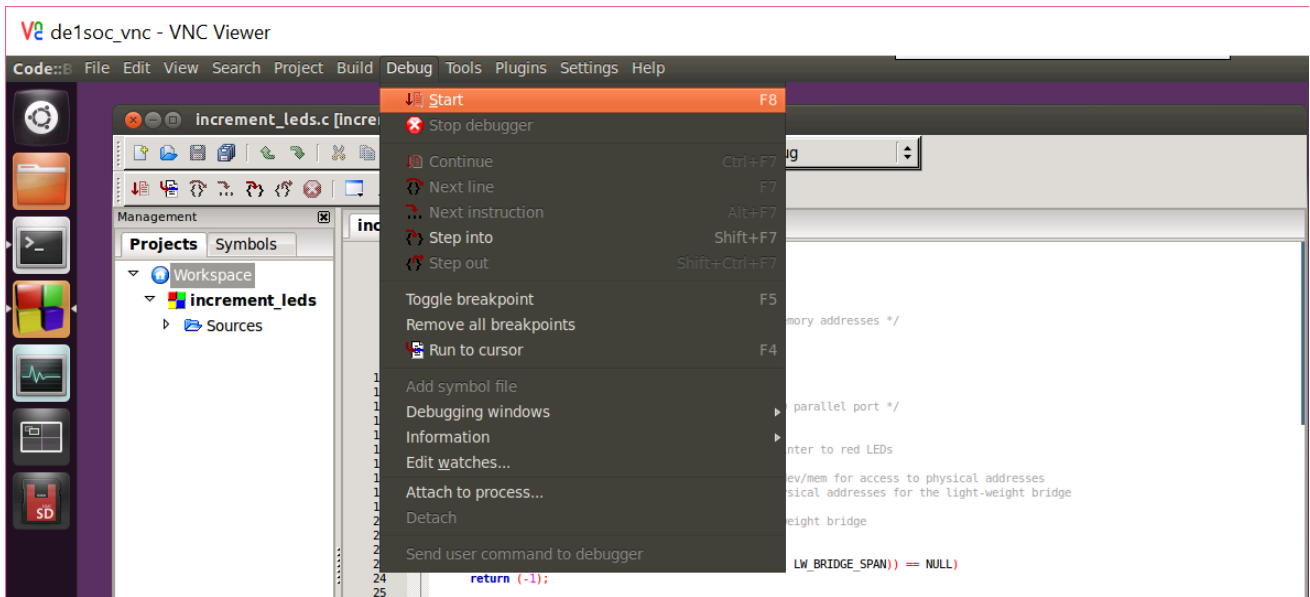


Figure 28. Starting the debugger.

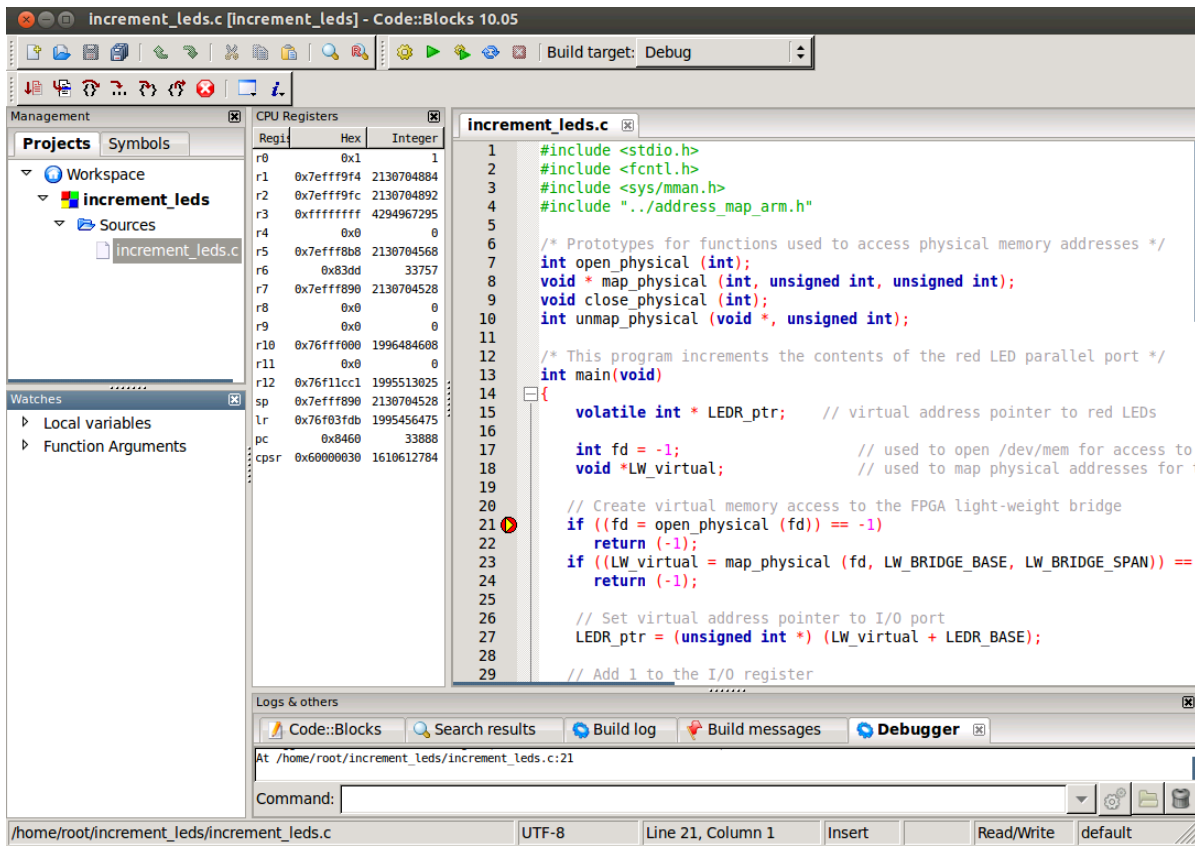


Figure 29. The debugging window.

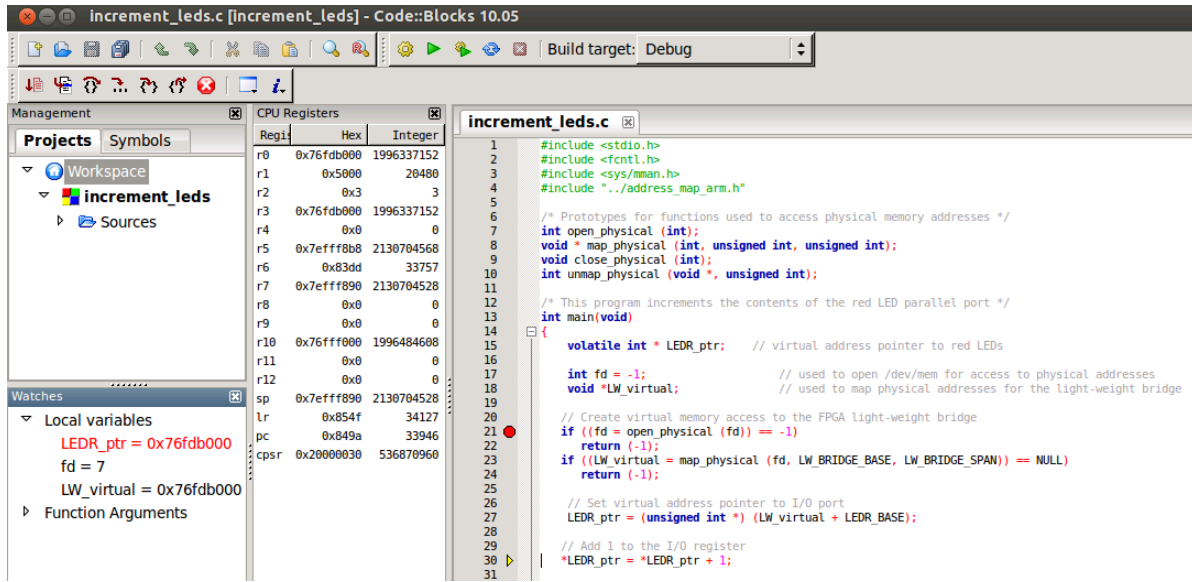


Figure 30. Displaying the values of variables.

Appendix B Include Files

Figure 31 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 31. The contents of the file *address_map_arm.h*.

Figure 32 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 32. The contents of the file *interrupt_ID.h*.

Appendix C Wrapper Functions for using Character Device Drivers with C Code

Header File for Pushbutton KEY Device

```
#ifndef KEYS_H
#define KEYS_H

/**
 * Function KEY_open: opens the pushbutton KEY device
 * Return: 1 on success, else 0
 */
int KEY_open (void);

/**
 * Function KEY_read: reads the pushbutton KEY device
 * Parameter data: pointer for returning data. If no KEYS are pressed *data = 0.
 * If all KEYS are pressed *data = 0b1111
 * Return: 1 on success, else 0
 */
int KEY_read (int * /*data*/);

/**
 * Function KEY_close: closes the KEY device
 * Return: void
 */
void KEY_close (void);

#endif
```

Header File for Slide Switch SW Device

```
#ifndef SW_H
#define SW_H

/**
 * Function SW_open: opens the slide switch SW device
 * Return: 1 on success, else 0
 */
int SW_open (void);

/**
 * Function SW_read: reads the SW device
 * Parameter data: pointer for returning data. If no switches are set *data = 0.
 * If all switches are set *data = 0b1111111111
 * Return: 1 on success, else 0
 */
int SW_read (int * /*data*/);

/**
 * Function SW_close: closes the SW device
 * Return: void
 */
void SW_close (void);

#endif
```

Header File for Red Light LEDR Device

```
#ifndef LEDR_H
#define LEDR_H

/**
 * Function LEDR_open: opens the red light LEDR device
 * Return: 1 on success, else 0
 */
int LEDR_open (void);

/**
 * Function LEDR_set: turns on/off the lights. If data = 0 all lights will be
 * turned off. If data = 0b1111111111 all lights will be turned on
 * Parameter data: written to LEDR device
 * Return: void
 */
void LEDR_set (int /*data*/);

/**
 * Function LEDR_close: closes the LEDR device
 * Return: void
 */
void LEDR_close (void);

#endif
```

Header File for Seven-Segment HEX Device

```
#ifndef HEX_H
#define HEX_H

/**
 * Function HEX_open: opens the 7-segment display HEX device
 * Return: 1 on success, else 0
 */
int HEX_open (void);

/**
 * Function HEX_set: sets the HEX displays to decimal digits from 0-9
 * Parameter data: the data to be displayed as a 6-digit decimal number. The upper
 * two digits will be displayed on HEX5-4, and the lower four digits on HEX3-0
 * Return: void
 */
void HEX_set (int /*data*/);

/**
 * Function HEX_raw: sets the HEX displays to any (raw) value
 * Parameter data_h: the lowest 16 bits are written to HEX5-4
 * Parameter data_l: a 32-bit value that is written to HEX3-0
 * Return: void
 */
void HEX_raw (int /*data_h*/, int /*data_l*/);

/**
 * Function HEX_close: closes the HEX device
 * Return: void
 */
void HEX_close (void);

#endif
```

Header File for VGA Video Device

```

#ifndef video_H
#define video_H

#define video_WHITE      0xFFFF    // Define some colors for video graphics
#define video_YELLOW    0xFFE0
#define video_RED       0xF800
#define video_GREEN     0x07E0
#define video_BLUE      0x041F
#define video_CYAN      0x07FF
#define video_MAGENTA   0xF81F
#define video_GREY      0xC618
#define video_PINK      0xFC18
#define video_ORANGE    0xFC00

/**
 * Function video_open: opens the VGA video device
 * Return: 1 on success, else 0
 */
int video_open (void);

/**
 * Function video_read: reads from the video device
 * Parameter cols: pointer for returning the number of columns in the display
 * Parameter rows: pointer for returning the number of rows in the display
 * Parameter tcols: pointer for returning the number of text columns
 * Parameter trows: pointer for returning the number of text rows
 * Return: 1 on success, else 0
 */
int video_read (int * /*cols*/, int * /*rows*/, int * /*tcols*/, int * /*trows*/);

/**
 * Function video_clear: clears all graphics from the video display
 * Return: void
 */
void video_clear (void);

/**
 * Function video_show: swaps the video front and back buffers
 * Return: void
 */
void video_show (void);

/**
 * Function video_pixel: sets pixel at (x, y) to color
 * Parameter x: the pixel column
 * Parameter y: the pixel row
 * Parameter color: the pixel color
 * Return: void
 */

```

```
void video_pixel (int /*x*/, int /*y*/, short /*color*/);

/**
 * Function video_line: draws a line
 * Parameter x1: the starting column
 * Parameter y1: the starting row
 * Parameter x2: the ending column
 * Parameter y2: the ending row
 * Parameter color: the line color
 * Return: void
 */
void video_line (int /*x1*/, int /*y1*/, int /*x2*/, int /*y2*/, short /*color*/);

/**
 * Draws a filled box on the VGA video device
 * Parameter x1: the column for one corner
 * Parameter y1: the row for one corner
 * Parameter x2: the column for the opposite corner
 * Parameter y2: the row for the opposite corner
 * Parameter color: the pixel color
 * Return: void
 */
void video_box (int /*x1*/, int /*y1*/, int /*x2*/, int /*y2*/, short /*color*/);

/**
 * Function video_text: puts text on the video device
 * Parameter x: the pixel column
 * Parameter y: the pixel row
 * Parameter msg: pointer to the text string
 * Return: void
 */
void video_text (int /*x*/, int /*y*/, char * /*string*/);

/**
 * Function video_erase: erases all text on the video device
 * Return: void
 */
void video_erase (void);

/**
 * Function video_close: closes the video device
 * Return: void
 */
void video_close (void);

#endif
```

Header File for Digital Audio Device

```

#ifndef audio_H
#define audio_H

#define MAX_VOLUME 0x7FFFFFFF // maximum audio sample value
#define MAX_SAMPLING_RATE 48000 // maximum audio sampling rate
#define MID_SAMPLING_RATE 32000 // medium audio sampling rate
#define MIN_SAMPLING_RATE 8000 // minimum audio sampling rate

/**
 * Function audio_open: opens the digital audio device
 * Return: 1 on success, else 0
 */
int audio_open (void);

/**
 * Function audio_read: reads data from the audio device
 * Parameter ldata: pointer for returning the left-channel data
 * Parameter rdata: pointer for returning the right-channel data
 * Return: 1 on success, else 0
 */
int audio_read (int * /*ldata*/, int * /*rdata*/);

/**
 * Function audio_init: initializes the digital audio device
 * Return: void
 */
void audio_init (void);

/**
 * Function audio_rate: sets the sampling rate of the digital audio device
 * Parameter data: the sampling rate in samples/sec (8000, 32000, or 48000)
 * Return: void
 */
void audio_rate(int /*data*/);

/**
 * Function audio_wait_write: waits for space to be available for writing
 * Return: void
 */
void audio_wait_write (void);

/**
 * Function audio_wait_read: waits until data is available for reading
 * Return: void
 */
void audio_wait_read (void);

/**
 * Function audio_write_left: writes data to the left channel

```



```
* Parameter data: left-channel data
* Return: void
*/
void audio_write_left (int /*data*/);

/**
 * Function audio_write_right: writes data to the right channel
 * Parameter data: right-channel data
 * Return: void
 */
void audio_write_right (int /*data*/);

/**
 * Function audio_close: closes the digital audio device
 * Return: void
 */
void audio_close (void);

#endif
```

Header File for 3-D Accelerometer Device

```

#ifndef accel_H
#define accel_H

/**
 * Opens the 3D-accelerometer accel device
 * Return: 1 on success, else 0
 */
int accel_open (void);

/**
 * Function: accel_read: reads data from the 3D-accelerometer accel device
 * Parameter ready: pointer for returning ready signal (1 if new data, else 0)
 * Parameter tap: pointer for returning the tap signal (1 if tap event, else 0)
 * Parameter dtap: pointer for the double-tap signal (1 if double-tap event,
 * else 0)
 * Parameter x: pointer for returning acceleration data in the x direction
 * Parameter y: pointer for returning acceleration data in the y direction
 * Parameter z: pointer for returning acceleration data in the z direction
 * Parameter mg_per_lsb: pointer for returning the acceleration-data scale factor
 * Return: 1 on success, else 0
 */
int accel_read (int * /*ready*/, int * /*tap*/, int * /*dtap*/, int * /*x*/,
    int * /*y*/, int * /*z*/, int * /*mg_per_lsb*/);

/**
 * Function: accel_init: initializes the 3D-acceleration device
 * Return: void
 */
void accel_init (void);

/**
 * Function accel_calibrate: xalibrates the 3D-acceleration device
 * Return: void
 */
void accel_calibrate (void);

/**
 * Function accel_device: request printing of the device ID
 * Return: void
 */
void accel_device (void);

/**
 * Functio accel_format: sets the format of acceleration data
 * Parameter full: a value of 1 sets full resolution
 * Parameter range: sets the G range, where G = {2, 4, 8, 16}
 * Return: void
 */
void accel_format (int /*full*/, int /*range*/);

```

```
/**
 * Function accel_rate: sets the data rate of acceleration data
 * Parameter rate: the data rate in Hz, where R = {25,12.5,6.25,1.56,0.78}
 * Return: void
 */
void accel_rate (float /*rate*/);

/**
 * Function accel_close: closes the 3-D acceleration device
 * Return: void
 */
void accel_close (void);

#endif
```

Appendix D Example C Program Using Device Drivers

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <intelfpgaup/KEY.h>
#include <intelfpgaup/video.h>

int screen_x, screen_y;
int char_x, char_y;

void gen_line (int *, int *, int *, int *, unsigned *);
void print_text (int, int, int, int, unsigned);

volatile sig_atomic_t stop;
void catchSIGINT(int signum){
    stop =1;
}

/* This program draws randomly-generated lines on the VGA display. The code uses
 * the character device drivers IntelFPGAUP/video and IntelFPGAUP/KEY. It draws a
 * random line each time a pushbutton KEY is pressed. The coordinates of each new
 * line are displayed on the bottom right of the screen. Exit by typing ^C. */
int main(int argc, char *argv[]) {
    int KEY_data;
    int x1, y1, x2, y2;
    unsigned color;
    char msg_buffer[80];
    time_t t;    // used to seed the rand() function

    // catch SIGINT from ^C, instead of having it abruptly close this program
    signal(SIGINT, catchSIGINT);

    srand ((unsigned) time(&t));    // seed the rand function
    // Open the character device drivers
    if (!KEY_open ( ) || !video_open ( ))
        return -1;

    video_read (&screen_x, &screen_y, &char_x, &char_y); // get screen & text size
    video_erase ( ); // erase any text on the screen

    // set random initial line coordinates
    gen_line (&x1, &y1, &x2, &y2, &color);

    /* There are two VGA buffers. The one we are drawing on is called the Back
     * buffer, and the one being displayed is called the Front buffer. The function
     * video_show swaps the two buffers, allowing us to display what has been drawn
     * on the Back buffer */

```

```

video_clear ( );    // clear current VGA Back buffer
video_show ( );    // swap Front/Back to display the cleared buffer
video_clear ( );    // clear the VGA Back buffer, where we will draw lines
video_line (x1, y1, x2, y2, color);
video_show ( );    // swap Front/Back to display the line
while (!stop) {
    sprintf (msg_buffer, "(%03d,%03d) (%03d,%03d) color:%04X", x1, y1, x2, y2,
            color);
    video_text (char_x - strlen(msg_buffer), char_y - 1, msg_buffer);
    printf ("Press a pushbutton KEY (^C to exit)\n");
    KEY_read (&KEY_data);
    while (!KEY_data && !stop)
        KEY_read (&KEY_data);
    gen_line (&x1, &y1, &x2, &y2, &color);
    video_show ( ); // swap Front/Back
    video_line (x1, y1, x2, y2, color);
    video_show ( ); // swap Front/Back to display lines
}
video_close ( );
KEY_close ( );
printf ("\nExiting sample program\n");
return 0;
}

/* Generate a new random line */
void gen_line (int *x1, int *y1, int *x2, int *y2, unsigned *color) {
    unsigned int video_color[] = {video_WHITE, video_YELLOW, video_RED,
        video_GREEN, video_BLUE, video_CYAN, video_MAGENTA, video_GREY,
        video_PINK, video_ORANGE};
    *x1 = rand()%(screen_x - 1);    // random x position
    *y1 = rand()%(screen_y - 1);    // random y position
    *x2 = rand()%(screen_x - 1);    // random x position
    *y2 = rand()%(screen_y - 1);    // random y position
    *color = video_color[(rand()%10)]; // random out of 10 video colors
}

```

Appendix E Wrapper Functions for using Character Device Drivers with Python* Code

Pushbutton KEY Device: KEY.py

```
def open_dev( ):
    ''' Opens the pushbutton KEY device

    :return: 1 on success, else 0
    '''

def read( ):
    ''' Reads the pushbutton KEY device

    :return: integer reflecting the KEY settings. If no KEYS are pressed
        returns 0. If all KEYS are pressed returns 0b1111
    '''

def close( ):
    ''' Closes the pushbutton KEY device

    :return: none
    '''
```

Slide Switch SW Device: SW.py

```
def open_dev( ):
    ''' Opens the slide switch SW device

    :return: 1 on success, else 0
    '''

def read( ):
    ''' Reads the slide switch SW device

    :return: integer reflecting the SW switch settings. If no switches are set
            returns 0. If all switches are set returns 0b1111111111
    '''

def close( ):
    ''' Closes the slide switch SW device

    :return: none
    '''
```

Red Light LEDR Device: LEDR.py

```
def open_dev( ):
    ''' Opens the red light LEDR device

    :return: 1 on success, else 0
    '''

def set(data):
    ''' Sets the red light LEDR device

    :param data: the integer data to write to LEDR. If data = 0 all lights will be
        turned off. If data = 0b1111111111 all lights will be turned on
    :return: none
    '''

def close( ):
    ''' Closes the red light LEDR device

    :return: none
    '''
```


Seven-Segment HEX Device: HEX.py

```
def open_dev( ):
    ''' Opens the 7-segment displays HEX device

    :return: 1 on success, else 0
    '''

def set(data):
    ''' Sets the HEX device in decimal number mode

    :param data: an integer to be displayed as a 6-digit decimal number. The upper
        two digits will be displayed on HEX5-4, and the lower four on HEX3-0
    :return: none
    '''

def raw(data1, data2):
    ''' Sets the HEX device in raw mode

    :param data1: an integer that is written to HEX5-4 as raw bits
    :param data2: an integer that is written to HEX3-0 as raw bits
    :return: none
    '''

def close( ):
    ''' Closes the HEX device

    :return: none
    '''
```

VGA Video Device: video.py

```
# define some graphics colors
WHITE, YELLOW, RED, GREEN, BLUE, CYAN, MAGENTA, GREY, PINK, ORANGE = \
    0xFFFF, 0xFFE0, 0xF800, 0x07E0, 0x041F, 0x07FF, 0xF81F, 0xC618, 0xFC18, 0xFC00

def open_dev( ):
    ''' Opens the VGA video device

    :return: 1 on success, else 0
    '''

def read( ):
    ''' Reads the video device

    :return: four integers: screen_x (x resolution), screen_y (y resolution),
            char_x (# of text columns), char_y (# of text lines)
    '''

def clear( ):
    ''' Erases all graphics in the current video frame buffer

    :return: none
    '''

def pixel(x, y, color):
    ''' Sets the pixel at coordinates (x, y) to color

    :param x: the column
    :param y: the row
    :param color: 16-bit VGA color
    :return: none
    '''

def line(x1, y1, x2, y2, color):
    ''' Draws a color line from graphics point (x1, y1) to (x2, y2)

    :param x1: the column for one end of the line
    :param y1: the row for one end of the line
    :param x2: the column for the other end of the line
    :param y2: the row for the other end of the line
    :param color: 16-bit VGA color
    :return: none
    '''

def box(x1, y1, x2, y2, color):
```

```
''' Draws a filled color box from corner (x1, y1) to corner (x2, y2)

:param x1: the column for one corner
:param y1: the row for one corner
:param x2: the column for the opposite corner
:param y2: the row for the opposite corner
:param color: 16-bit VGA color
:return: none
'''

def text(x, y, msg):
    ''' Draws the text msg at character coordinates (x, y)

    :param x: the character column
    :param y: the character row
    :param msg: the text string
    :return: none
    '''

def erase( ):
    ''' Erases all text in the video character buffer

    :return: none
    '''

def show( ):
    ''' Swaps the front/back video frame buffers

    :return: none
    '''

def close( ):
    ''' Closes the video device

    :return: none
    '''
```

Digital Audio Device: audio.py

```
MAX_VOLUME = 0x7FFFFFFF
MAX_SAMPLING_RATE, MID_SAMPLING_RATE, MIN_SAMPLING_RATE = 48000, 32000, 8000

def open_dev( ):
    """ Opens the digital audio device

    :return: 1 on success, else 0
    """

def read( ):
    """ Reads the audio device

    :return: two integers: left-channel data, right-channel data
    """

def init( ):
    """ Initializes the audio device

    :return: none
    """

def sampling_rate(data):
    """ Sets the audio device sampling rate

    :param data: sampling rate in thousands/sec., where data = {8000, 32000, 48000}
    :return: none
    """

def wait_write( ):
    """ Waits for space to be available for writing to the audio device

    :return: none
    """

def wait_read( ):
    """ Waits until data is available for reading from the audio device

    :return: none
    """

def write_left(data):
    """ Writes data to the left audio channel
```

```
:param data: integer value to be written  
:return: none  
'''
```

```
def write_right(data):  
    ''' Writes data to the right audio channel  
  
    :param data: integer value to be written  
    :return: none  
    '''
```

```
def close( ):  
    ''' Closes the audio device  
  
    :return: none  
    '''
```

3-D Accelerometer Device: accel.py

```
def open_dev( ):
    ''' Opens the 3D-accelerometer accel device

    :return: 1 on success, else 0
    '''

def read( ):
    ''' Reads the accel device

    :return: seven integers: ready (1 if new acceleration data is available, else 0),
        tap (1 if tap event, else 0), dtap (1 if double-tap event, else 0),
        x (acceleration in the x axis), y (... y axis), z (... z axis),
        scale (mG per lsb scale factor for acceleration data)
    '''

def init( ):
    ''' Initializes the 3D-acceleration device

    :return: none
    '''

def calibrate( ):
    ''' Calibrates the 3D-acceleration device

    :return: none
    '''

def device( ):
    ''' Request printing of the device ID from the 3D-acceleration device

    :return: none
    '''

def format(full, range):
    ''' Sets the format of acceleration data

    :param full: integer value of 1 sets full resolution
    :param range: integer to set the G range to {2, 4, 8, 16}
    :return: none
    '''

def rate(rate):
    ''' Sets the data rate of acceleration data
```

```
:param rate: float value to set rate to {25,12.5,6.25,1.56,0.78} Hz  
:return: none  
'''
```

```
def close( ):  
''' Closes the accel device  
  
:return: none  
'''
```

Appendix F Example Python* Program Using Device Drivers

```
''' This program draws randomly-generated lines on the VGA display

The code uses the character device drivers IntelFPGAUP/video and IntelFPGAUP/KEY.
It draws a random line each time a pushbutton KEY is pressed. The coordinates of
each new line are displayed on the bottom right of the screen. Exit by typing ^C
'''

import sys
import KEY
import video
import signal
import random
from random import randint

video_color = [video.WHITE, video.YELLOW, video.RED, video.GREEN, video.BLUE,
               video.CYAN, video.MAGENTA, video.GREY, video.PINK, video.ORANGE]

stop = False

def signal_handler(signal, frame) :
    global stop
    stop = True

def gen_line () :
    ''' generate a random line with two (x,y) coordinates and a color

    return: the x,y coordinates and color
    '''
    x1 = randint(0,screen_x-1)
    y1 = randint(0, screen_y-1)
    x2 = randint(0,screen_x-1)
    y2 = randint(0, screen_y-1)
    color = random.choice(video_color)
    return x1, y1, x2, y2, color

signal.signal(signal.SIGINT, signal_handler) # exit cleanly on ^C
if not KEY.open_dev() or not video.open_dev():
    sys.exit()

screen_x, screen_y, char_x, char_y = video.read() # get VGA graphics and text size
video.erase() # erase any text on the screen

# generate a random line
x1, y1, x2, y2, color = gen_line()

# There are two VGA buffers. The one we are drawing on is called the Back buffer;
# the one being displayed is called the Front buffer. The function video_show swaps
# the two buffers, allowing us to display what has been drawn on the Back buffer
video.clear() # clear current VGA Back buffer
video.show() # swap Front/Back to display the cleared buffer
```



```
video.clear()    # clear the current VGA Back buffer, where we will draw lines
video.line(x1, y1, x2, y2, color)
video.show()    # swap Front/Back to display the line

while not (stop) :
    msg = "(%03d,%03d) (%03d,%03d) color:%04X" % (x1, y1, x2, y2, color)
    video.text(char_x - len(msg), char_y - 1, msg)
    print("Press a pushbutton KEY (^C to exit)\n")
    while (KEY.read() == 0) and not (stop) :
        pass    # wait
    x1, y1, x2, y2, color = gen_line()
    video.show()    # swap Front/Back
    video.line(x1, y1, x2, y2, color)
    video.show()    # swap Front/Back to display lines

KEY.close()
video.close()
print("Exiting sample program\n")
```

Appendix G 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 your DE-series board, 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-eabihf-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 33. 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 33. The helloworld program

As mentioned, we will be using the `arm-linux-gnueabihf-` 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 34. Navigate to the folder that contains the `helloworld.c` file by using the `cd` command.

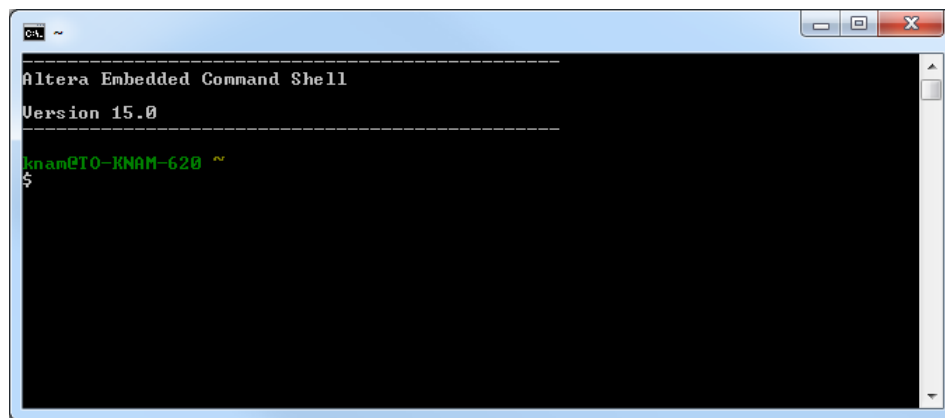
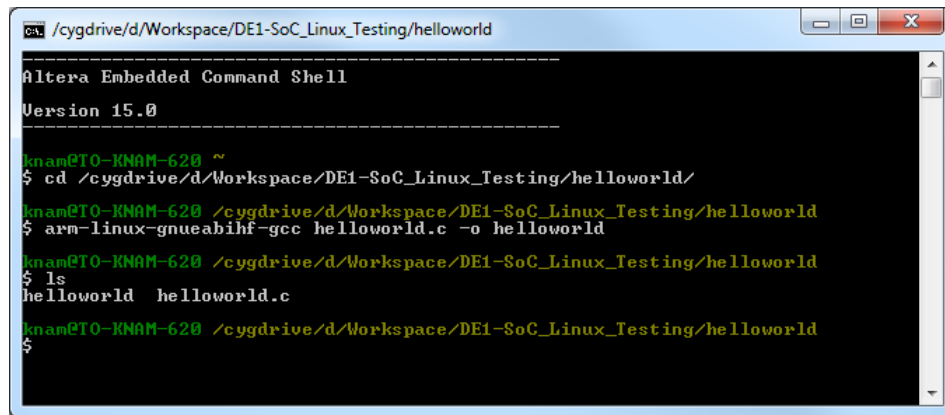


Figure 34. The Embedded Command Shell

Compile the code using the command `arm-linux-gnueabihf-gcc helloworld.c -o helloworld`, as shown in Figure 35. 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 your DE-series board. To copy the executable file you can either use an *ftp* program as discussed in Section 2.7.4, or another method of your choosing.



```
ca: /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-gnueabi-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 35. Cross-compiling the helloworld program

In addition to `arm-linux-gnueabi-gcc`, the `arm-linux-gnueabi-` 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. If you have a network connection between your host computer and DE-series board, then an easy way to transfer files is to use the FTP server, as discussed in Section 2.7.4. Otherwise, you can copy files directly onto your microSD card using the procedures described below. 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 your DE-series 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). *Never* delete these files, as they are 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). *Never* delete these files, as they are required to boot Linux.

Appendix H 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 *DE1-SoC-UP* Linux distribution contains drivers for this mechanism, allowing us to program the FPGA from the Terminal window. 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 37. Enable file compression by ticking the checkbox as shown, then press OK.
7. We are now ready to generate the .rbf file. Click `Generate`. You should see the success message shown in Figure 38.
8. Finally, we can transfer the .rbf file to the Linux file system, using a method such as ftp.

Programming the FPGA

While Linux 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 file `/home/root/DE1_SoC_Computer.rbf`. If you examine the contents of the `programfpga` script, you will see that it comprises the 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
dd if=/home/root/DE1_SoC_Computer.rbf of=/dev/fpga0 bs=1M
echo 1 > /sys/class/fpga-bridge/fpga2hps/enable
echo 1 > /sys/class/fpga-bridge/hps2fpga/enable
echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable
```

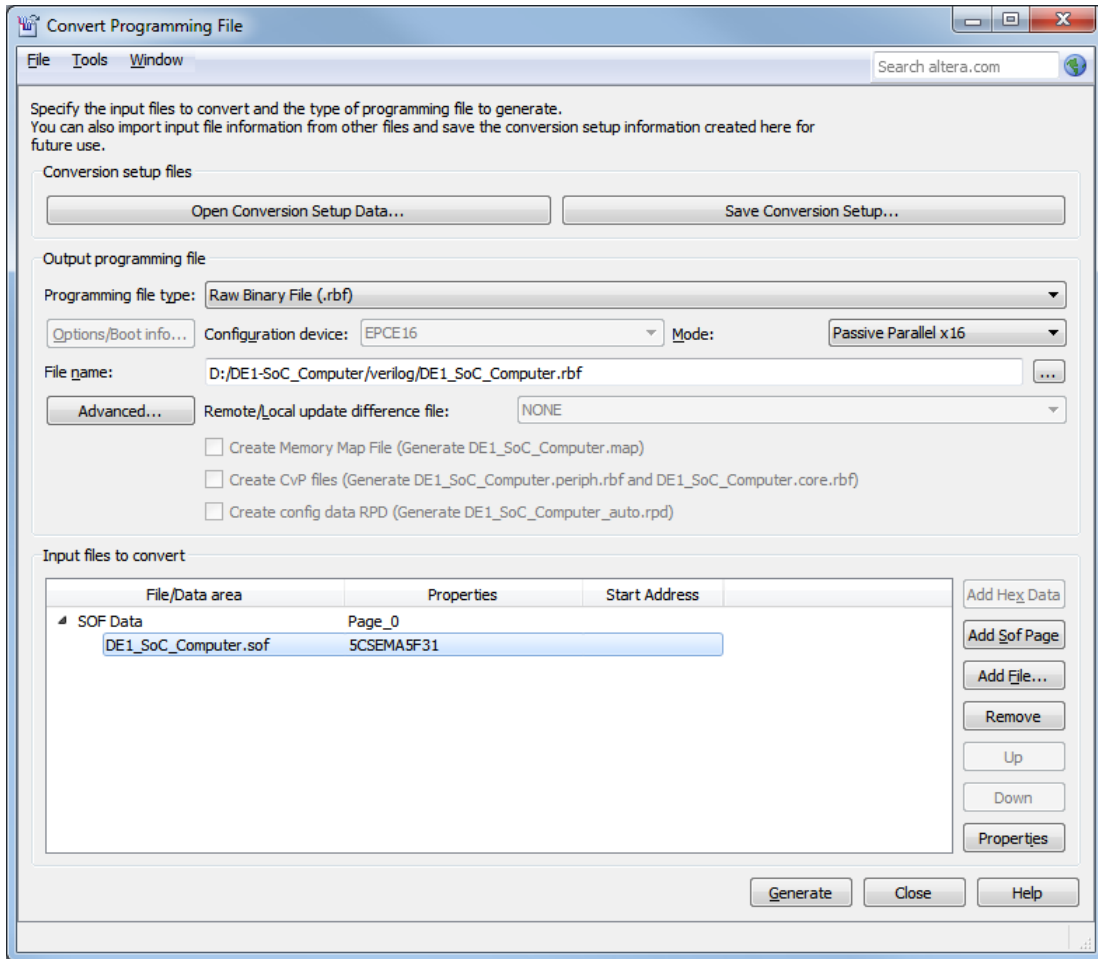


Figure 36. The Convert Programming File Tool.

The *.rbf* file is loaded into the FPGA device using the command

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

where the input *<filename>* is `/home/root/DE1_SoC_Computer.rbf`. To program a different file into the FPGA you need to execute the above commands, and specify whatever *.rbf* file is wanted.

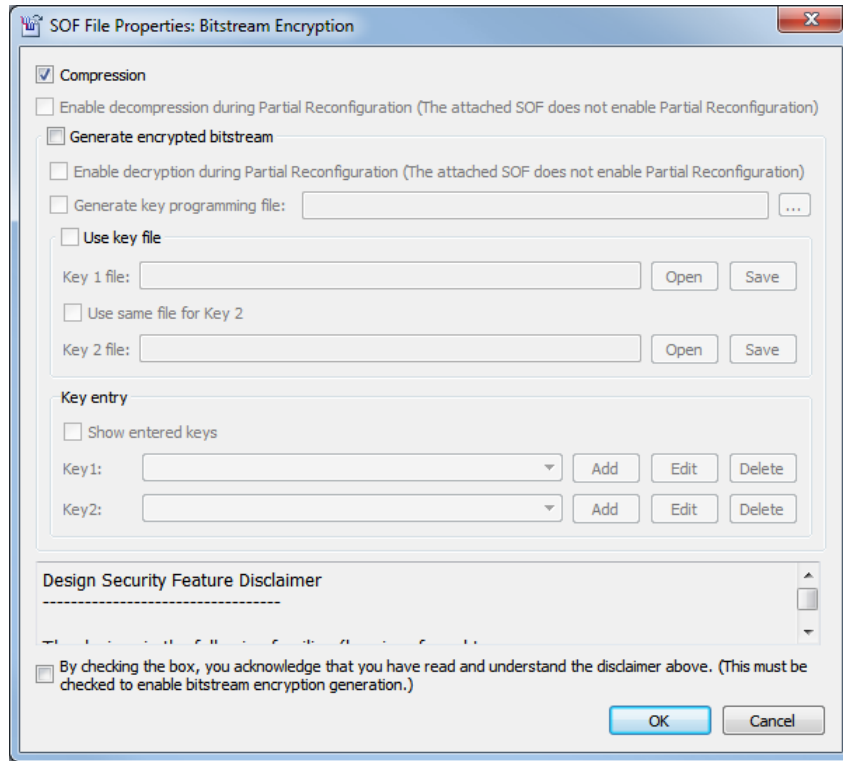


Figure 37. Enabling file compression.

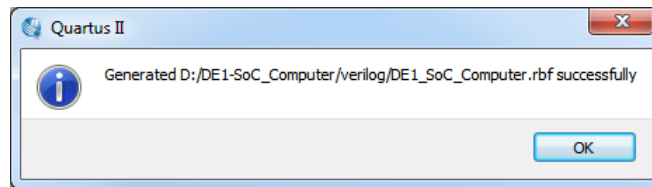


Figure 38. The .rbf file successfully generated.

Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.