Using Linux on the DE1-SoC

For Quartus Prime 16.0

1 Introduction

This tutorial describes a Linux operating system that runs on the DE1-SoC Computer. This computer system is implemented on the Altera DE1-SoC development and education board. Linux runs on the ARM Cortex-A9 dual-core processor that is part of the Cyclone V SoC device on the DE1-SoC board. The tutorial shows how Linux can be programmed onto a microSD memory card and booted by the ARM processor. We also show how software programs can be developed that run on the ARM processor under Linux, and that can make use of the hardware resources in the DE1-SoC Computer. These resources include peripherals in the hard processor system (HPS), and custom hardware peripherals implemented in the FPGA, in the Cyclone V SoC device.

Contents:

- Getting Started with Linux on the DE1-SoC board
- Configuring the FPGA from Linux
- Developing Linux Applications that use FPGA Hardware Devices
- Developing Linux Drivers for FPGA Hardware

Requirements:

- Altera DE1-SoC development and education board
- Host computer, running either Microsoft Windows (version 10 is recommended) or Linux (Ubuntu, or a similar Linux distribution). The host computer is used for developing software programs that run under Linux on the DE1-SoC board
- Ethernet cable, and/or Mini-USB cable, for connecting the DE1-SoC board to the host computer
- MicroSD card (8GB or larger)

Optional:

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

2 Running Linux on the DE1-SoC Board

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

2.1 The Cyclone V SoC Device

The DE1-SoC board features a Cyclone V SoC chip, which contains two main components: the Hard Processor System (HPS), and the FPGA. The HPS contains an ARM Cortex-A9 processor (which we will use to run Linux), and various peripheral devices such as timers, GPIO, USB, and Ethernet. A feature of the SoC is that 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 DE1-SoC-UP Linux Distribution Image

Altera and Terasic Technologies provide a number of Linux distributions that you can use to quickly get Linux running on the DE1-SoC board. These Linux distributions range from a simple command line only version, to the more full-featured Ubuntu Linux distribution with a GUI interface. The Linux distributions are provided in the *.img* (image) file format, which can be written to a microSD card and booted on the DE1-SoC board. For this tutorial we will use the *DE1-SoC-UP* Linux distribution, whose image file *DE1-SoC-UP-Linux.img* can be downloaded from the Altera University Program website.

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 and C++ programs. We will make extensive use of this toolchain to compile programs in Section 3. Another feature is the automatic programming of the FPGA as *DE1-SoC-UP* starts up. The FPGA is programmed with the *DE1-SoC Computer* system, which contains IP cores that communicate with the peripheral devices found on the DE1-SoC board, such as the switches, LEDs, pushbuttons, VGA, and audio. In Section 3.3, we will show how to write programs that communicates with Parallel IO (PIO) cores of the *DE1-SoC Computer* to access the LEDs and pushbuttons of the board. The *DE1-SoC Computer* system is described in detail in the document *DE1-SoC Computer with ARM*.

2.3 Preparing the Linux MicroSD Card

The DE1-SoC board is designed to boot Linux from an inserted Linux microSD card. In this section, you will prepare a Linux microSD card by placing the *DE1-SoC-UP-Linux.img* image file into a microSD card (8 GB or larger). This section of the tutorial assumes that you have access to a computer with a microSD card reader. 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 the microSD card into your computer using a microSD card reader, then launch Win32 Disk Imager.
2. Select the drive letter corresponding to the microSD card under `Device`, as shown in Figure 1.

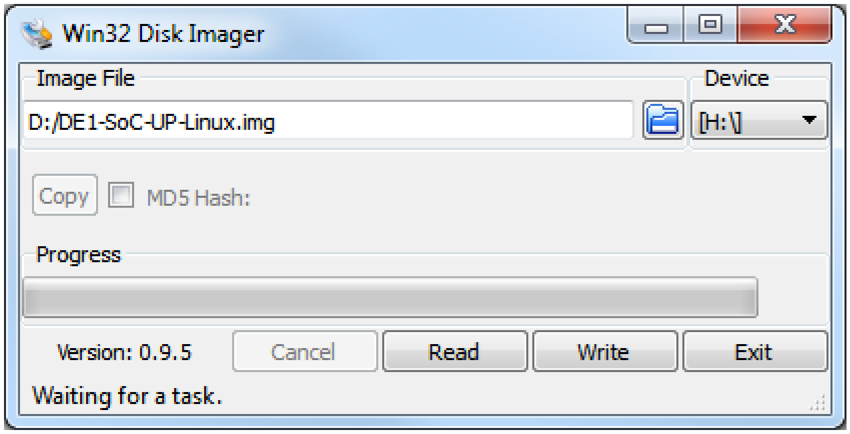

Figure 1. The Win32 Disk Imager tool.

3. Select the *DE1-SoC-UP-Linux.img* image under `Image File`, as shown in Figure 1. This image can be found on the Altera 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.

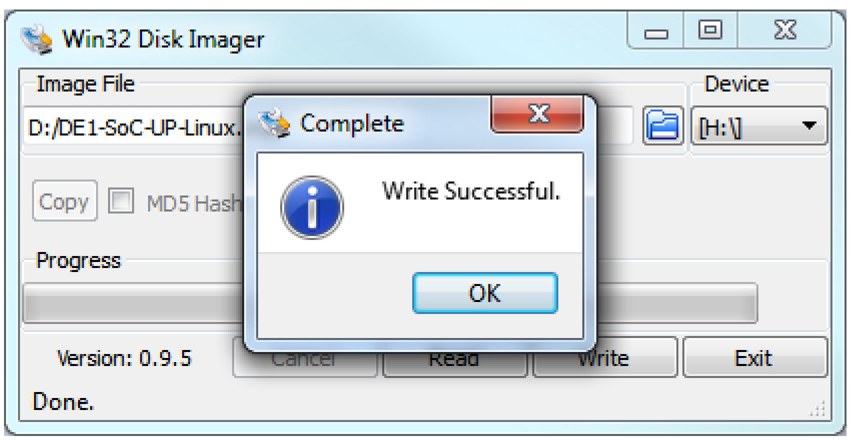

Figure 2. Writing a file to the microSD card using Win32 Disk Imager.

2.4 Configuring the DE1-SoC Board for use with Linux

First ensure that the DE1-SoC board is powered off, and then insert the Linux microSD card into the microSD card slot. Before turning on the board, configure the MODE SELECT (MSEL) switches found on the underside of the board to match the settings shown in Figure 3. This setting configures the Cyclone V SoC chip so as to allow the ARM processor to program the FPGA (it is necessary to make this setting because our Linux image programs the FPGA as part of its boot-up process). In addition, we will need this MSEL configuration in Appendix D, where we show how to manually program the FPGA from the Linux command line.

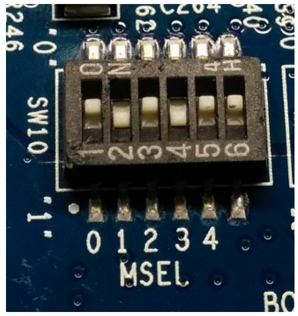

Figure 3. Configuring the MSEL switches of the DE1-SoC board.

2.5 Connecting the DE1-SoC Board to the Host Computer

Before booting Linux, you should first connect the DE1-SoC board to your host computer. There are two main methods of communicating between the DE1-SoC board and the host computer: using a USB cable to connect to a Linux command-line prompt, or using a network connection to connect to a Linux graphical user interface (GUI). Each method is described below.

2.6 Connecting to the Host Computer using a USB Cable

The DE1-SoC-UP Linux image has been configured to send and receive text (on the standard streams stdout, stdin, stderr) via the Cyclone V HPS's UART. This UART is a device that facilitates serial communication of characters. On the DE1-SoC board, the HPS's UART is attached to a UART-to-USB chip that can be connected to a host computer by using a USB cable. On the host computer, we can use a *terminal* program to display this text. Various *terminal* programs are available via the internet. For this tutorial, we will be using the free-to-use tool `Putty` which is available for both Windows and Linux.

In the following discussion we assume that you have installed Putty on your Host Computer. If you choose to install and use a different terminal program, then the instructions below would need to be modified accordingly. Connect the UART-to-USB port of the DE1-SoC board to your host computer using the mini-USB cable supplied with the board. The UART-to-USB connector can be found immediately next to the microSD card slot. If this is your first time connecting to the UART-to-USB chip, you may have to install its device driver on your host computer. If your host operating system does not automatically install the driver, then you can search for it on the Internet.

2.6.1 Using a Windows Host Computer

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

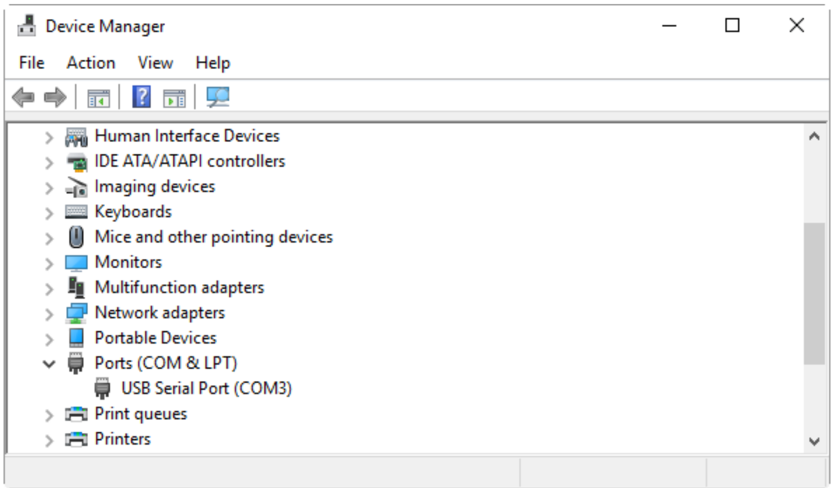

Figure 4. Determining the COM port of the UART-to-USB connection in Device Manager.

2.6.2 Using a Linux Host Computer

On a Linux 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 PC, each TTY device is assigned a unique identifier. The name assigned to your UART-to-USB connection can be determined by running the command `dmesg | grep tty` as shown in Figure 5. In 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.

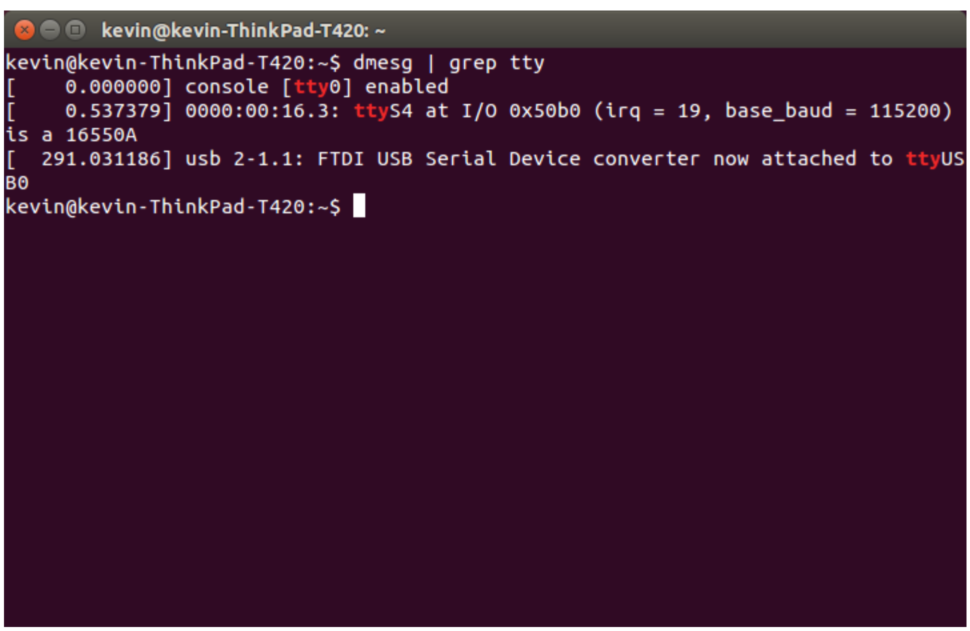

Figure 5. Determining the TTY device that corresponds to the UART-to-USB connection.

Start the Putty program. Now that the serial device (COM port or TTY device) corresponding to the UART-to-USB 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.

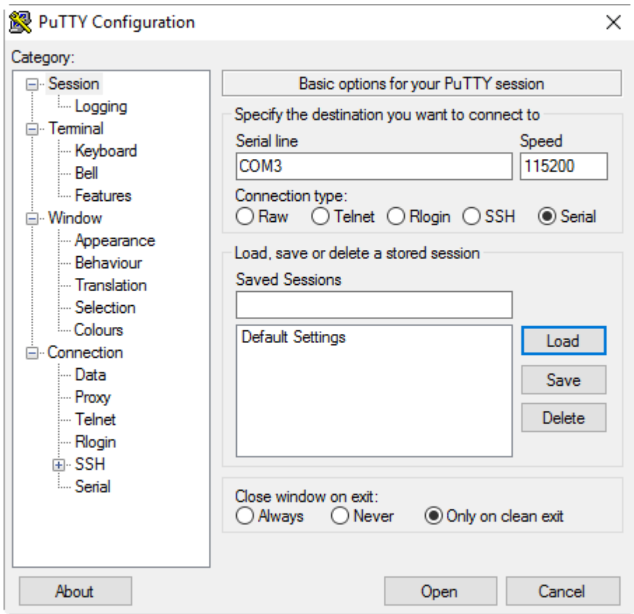

Figure 6. Putty's main window.

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 8, stop bits to 1, and parity and flow control to *none*.

Once all of the serial-line settings have been entered, press `Open` to start the terminal. Now, turn on the power to the DE1-SoC 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 Figure 8. Once Linux has finished booting, you will be logged in to the Linux command line interface (CLI) as the *root* user. Being logged in as root means that you that have administrator-level privileges, which allow you to modify settings and execute privileged programs.

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.

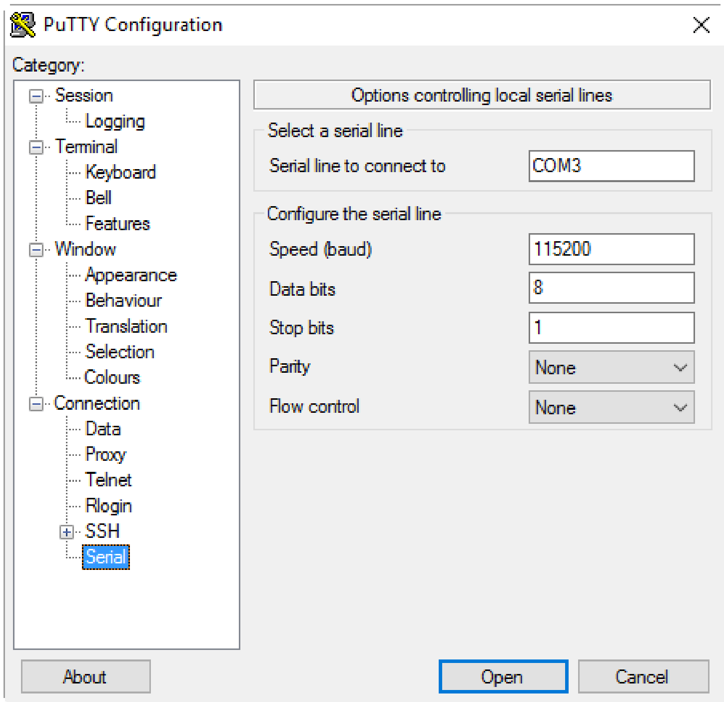

Figure 7. Putty's configuration window for serial communication settings.

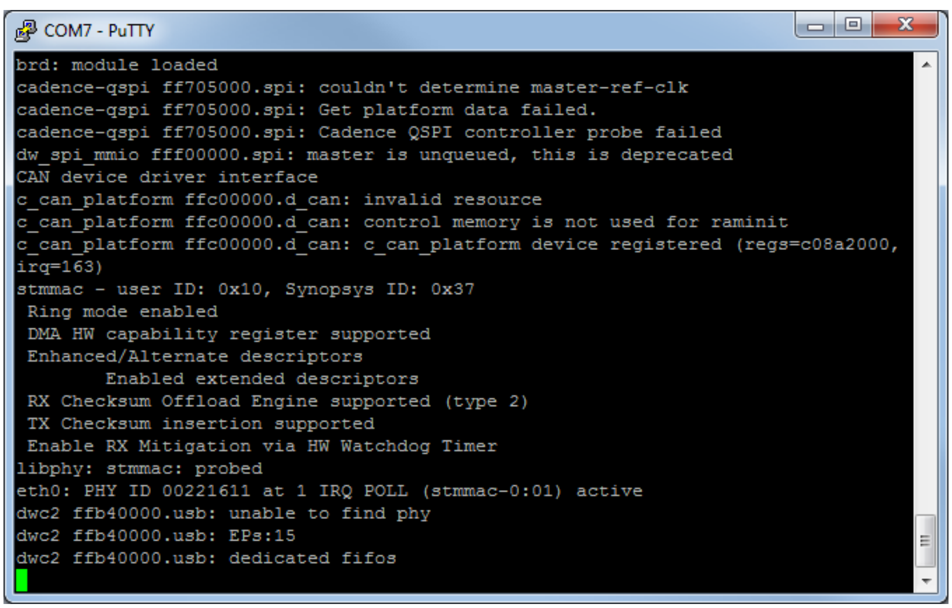

Figure 8. Putty terminal displaying text output as the Linux kernel boots up.

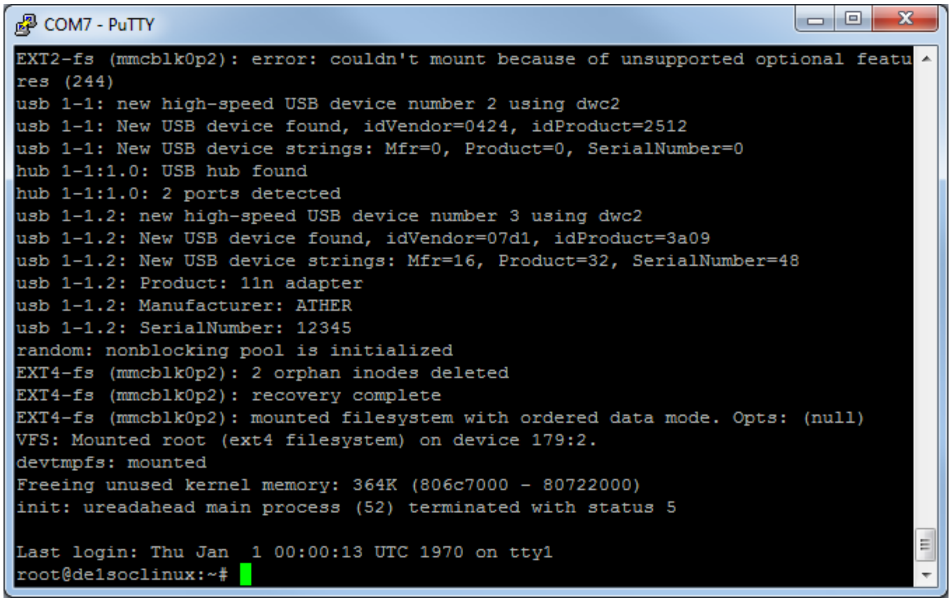

Figure 9. The Linux command line prompt showing the root ('#') logon.

2.7 Connecting to the Host Computer using a Network

The DE1-SoC-UP 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 DE1-SoC 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 DE1-SoC board: using an Ethernet cable, and using a WiFi adapter. Both methods are discussed below.

2.7.1 Connection using an Ethernet Cable

To establish a network connection to your host computer using an Ethernet cable, plug one end of the Ethernet cable into the RJ45 port on the DE1-SoC 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 DE1-SoC 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 DE1-SoC 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 DE1-SoC board. If your host computer is currently connected to the Internet, and you wish to maintain this Internet connection, then the best option is to change the IP address of the DE1-SoC board so that it uses the same subnet as the host computer. But if you do not need to access the Internet on the host computer, then you may wish to instead change the IP address of your host computer to use the same subnet as the DE1-SoC board.

The procedures for changing the IP address of the host computer or DE1-SoC board are described below.

Changing the IP address of your Host Computer

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

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

If you are using Linux on the host computer, the IP address can be changed by using the `ifconfig` command. If your Ethernet adapter were called *eth0*, then the command would be `ifconfig eth0 192.168.0.`*xxx* (or `192.168.1.`*xxx*), where *xxx* is a number of your choosing (that is not already being used in the subnet). If your Ethernet adapter is not called *eth0*, then replace this field with the actual name of your Ethernet port.

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

Changing the IP address of the DE1-SoC Board

To change the IP address of the DE1-SoC 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 DE1-SoC board execute the `ifconfig` command. For example, if your host computer's IP address were 169.254.245.156, then you would use the command `ifconfig eth0 169.254.245.xxx`, where `xxx` is a number of your choosing (that is not already being used in the subnet).

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

2.7.2 Connecting to the Host Computer using a WiFi Adapter

To make a network connection to the DE1-SoC board using a WiFi adapter, you first have to connect your host computer to the board via a USB cable, as described in section 2.6. Then, you can use a Terminal window connected to Linux on the DE1-SoC board to connect to the desired WiFi network.

The DE1-SoC board 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 useable, but drivers may need to be manually installed.

Plug your WiFi adapter into a USB port on the DE1-SoC board. To join a desired WiFi network, you can run the following script: `connect_wpa <ssid> <password>.` This script can be found in */home/root/misc*. The DE1-SoC board should become connected to your WiFi network after a few moments.

Instead of using the `connect_wpa` script, it is possible to run the Linux commands included in the script manually. First, using the Terminal window connected to Linux on the DE1-SoC 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 DE1-SoC 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 to connect to the VNC server as described in Section 2.7.3.

2.7.3 Using the VNC Server

After you have set up a network connection between your host computer and the DE1-SoC 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. 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 window and typed the Linux command `pwd`.

The VNC Server on the DE1-SoC Computer supports four different screen sizes. In the Terminal window the screen size can be changed by using the `xrandr` command. Type `xrandr -s 0` to set the smallest screen size, and `xrandr -s 3` to choose the largest size.

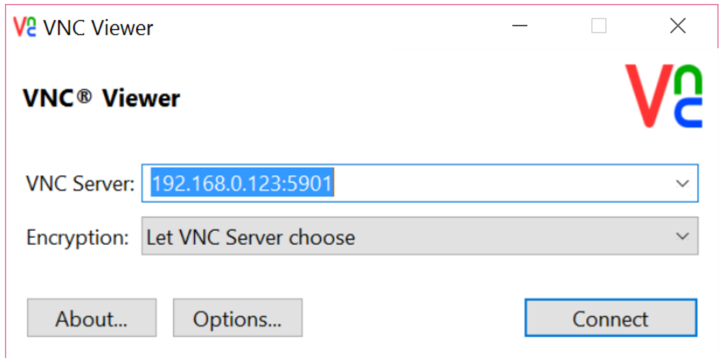

Figure 10. The RealVNC opening dialog.

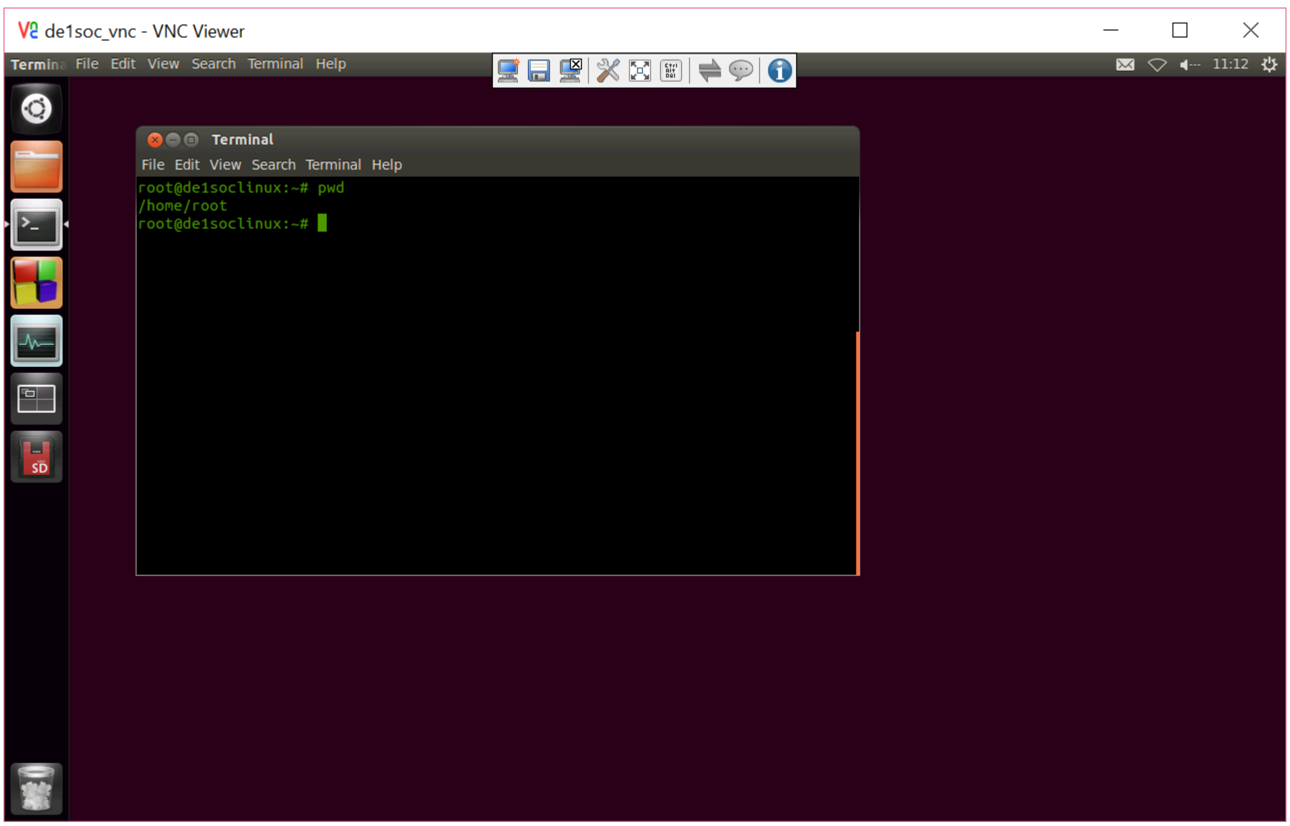

Figure 11. The main RealVNC window.

The DE1-SoC-UP 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.

2.7.4 Transferring Files to/from the Host Computer

The DE1-SoC-UP Linux image includes an *FTP* Server that can be used to transfer files between the host computer and the DE1-SoC Computer. The FTP server uses the secure FTP (SFTP) protocol and uses *Port 22* of the DE1-SoC network connection. The login *Username* for the FTP Server is *root*, and the password is *password*. An easy-to-use FTP client for both Windows and Linux host computers is *FileZilla*, available from *https://filezilla-project.org/*.

2.7.5 Accessing the Internet

After connecting the DE1-SoC Computer to a network, it is possible to provide Internet access to Linux applications. For example, the DE1-SoC-UP 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 the DE1-SoC Board

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

3.1 Native Compilation on the DE1-SoC Board

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

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

```c
#include <stdio.h>

int main(void){

    printf("Hello World!\n");

    return 0;
}
```

Figure 12. The helloworld program

Open a command-line interface as described in Section 2.6, or open a Terminal window in the VNC graphical interface as discussed in Section 2.7. Change your working directory to */home/root/helloworld*. Compile the program using the command `gcc helloworld.c -o helloworld`, as shown in Figure 13. The `gcc` command invokes the *GNU C Compiler*, which is an open-source compiler that is widely used to compile Linux programs. In our `gcc` command, we supply two arguments. The first is the source code file, `helloworld.c`, which contains our code. The second is `-o helloworld` which tells the compiler to output an executable file named *helloworld*. Once the compilation is complete, we can run the program by typing `./helloworld`. The program outputs the message "Hello World!" then exits, as shown in Figure 13.

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

3.2 Accessing Hardware Devices in the FPGA from a Linux Program

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

If we were developing a baremetal ARM program (a program that does not run on top of an operating system), then accessing peripherals in the FPGA that are mapped to a memory region would be done by simply reading from, or writing to, the appropriate memory address. Examples of software programs that access memory-mapped peripherals in the FPGA can be found on the Altera 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 access the characters in the text file by reading the virtual memory address span to which the file has been mapped. The system memory device file, */dev/mem*, is a file that represents the physical memory of the computer system. An access into this file at some offset is equivalent to accessing physical memory at the offset address. By using `mmap` to map the */dev/mem* file into virtual memory, we can map physical addresses to virtual addresses, allowing programs to access physical addresses. In the following section, we will examine a sample Linux program that uses `mmap` and */dev/mem* to access the Lightweight HPS-to-FPGA (*lwhps2fpga*) bridge's memory span and communicate with an IP core on the FPGA.

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 uses the *lwhps2fpga* bridge to alter the state of the red LEDs on the DE1-SoC board. Each time this program is executed, the value displayed on the red LEDs incremented by one. The code is shown in Figure 14.

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

The example code can be found at */home/root/increment_leds/increment_leds.c*. You can compile the code using the command `gcc increment_leds.c -o increment_leds`. Important lines of the code are described below.

- Lines 3-4 include the `fcntl.h` and `sys/mman.h` header files, which are needed to use */dev/mem* device file and the *mmap* and *munmap* kernel functions.
- Line 5 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 8-11 provide prototype declarations for functions that are used to access physical memory. These functions are listed in Figure 15. The functions `open_physical` and `close_physical` are used to open/close the */dev/mem* device file. The function *map_physical* calls the `mmap` kernel function to create a physical-to-virtual address mapping for I/O devices, and the `unmap_physical` closes this mapping. These four functions can be used in any program that needs to access physical memory addresses, along with the address information given in Appendix B.

- Line 22 opens the file */dev/mem*
- Line 24 maps a part of the */dev/mem* file into memory. It maps a portion that starts at the base address of *lwhps2fpga* (`LW_BRIDGE_BASE`) and spans `LW_BRIDGE_SPAN` bytes. The `LW_virtual` variable will be set to an address that maps to the bottom of the requested physical address space (`LW_BRIDGE_BASE`). This means an access to `LW_virtual + offset` will access the physical address `0xFF200000 + offset`.
- Line 28 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 31 reads the *data* register of the LED port, increments the value by one, then writes the incremented value back to the register.
- Lines 33-34 unmap and close the */dev/mem* file

```
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
17    int fd = -1;                // used to open /dev/mem
18    void *LW_virtual;           // physical addresses for light-weight bridge
19
20    // Create virtual memory access to the FPGA light-weight bridge
21    if ((fd = open_physical (fd)) == -1)
22       return (-1);
23    if ((LW_virtual = map_physical (fd, LW_BRIDGE_BASE, LW_BRIDGE_SPAN)) ==
         NULL)
24       return (-1);
25
26    // Set virtual address pointer to I/O port
27    LEDR_ptr = (unsigned int *) (LW_virtual + LEDR_BASE);
28
29    // Add 1 to the I/O register
30    *LEDR_ptr = *LEDR_ptr + 1;
31
32    unmap_physical (LW_virtual, LW_BRIDGE_SPAN);
33    close_physical (fd);
34    return 0;
35 }
```

Figure 14. C-code for the *increment_leds* program

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

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

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

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

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

Figure 15. Functions for managing physical memory addresses.

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; adding 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 KEYs port in the DE1-SoC Computer. To make the example more interesting, we use ARM processor interrupts to handle KEY presses. 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 Interupt Controller* (GIC) which can accommodate 256 interrupt request (IRQ) lines ranging from *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 KEYs 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 Linux to register an interrupt handler. The Linux header file `linux/interrupt.h` provides this interface, among which is the function `request_irq(...)`. This function takes an integer argument `irq` and a function pointer argument `handler`, and registers the function as the handler for IRQ number `irq`.

3.4.1 The Pushbutton Interrupt Handler Kernel Module

The code for our kernel module is shown in Figure 16. Unlike code for normal programs, the kernel module code has no `main` function. Instead, every kernel module has an `init` function which is executed when the module is inserted into the kernel, and an `exit` function which is executed when the module is removed from the kernel. These functions are specified using the macros `module_init(...)` and `module_exit(...)`.

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

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

In this example, `irq_handler(...)` serves as a trivial example of an interrupt handler. A "real" driver for a device would do something more useful like transfer data to and from buffers, check the status of devices, and the like. A device driver that does not use interrupts would still look similar to the code in Figure 16, 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.

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

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

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

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

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

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

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

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

Figure 16. C-code for the pushbutton interrupt handler kernel module

3.4.2 Compiling the Kernel Module

The kernel module source code can be found in the directory */home/root/pushbutton_irq_handler/*. To compile the module, use the included Makefile by running the Linux command `sf make`. The contents of the Makefile are shown in Figure 17. The first line, `obj-m += <module_name>.o`, specifies the name of the kernel module that is to be built (our kernel module will as a result be named *pushbutton_irq_handler*). This line also tells the build system to look for the kernel module code in <module_name>.c, and generate the kernel object file <module_name>.ko at the end of the compilation.

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

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

```
obj-m += pushbutton_irq_handler.o

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

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

Figure 17. Kernel module makefile

3.4.3 Running the Kernel Module

A kernel module is executed by *inserting* it into the Linux kernel using the command `insmod` <*filename*`.ko`>. Insert the kernel module you compiled above by using the command `insmod pushbutton_irq_handler.ko`, as shown in Figure 18. 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.

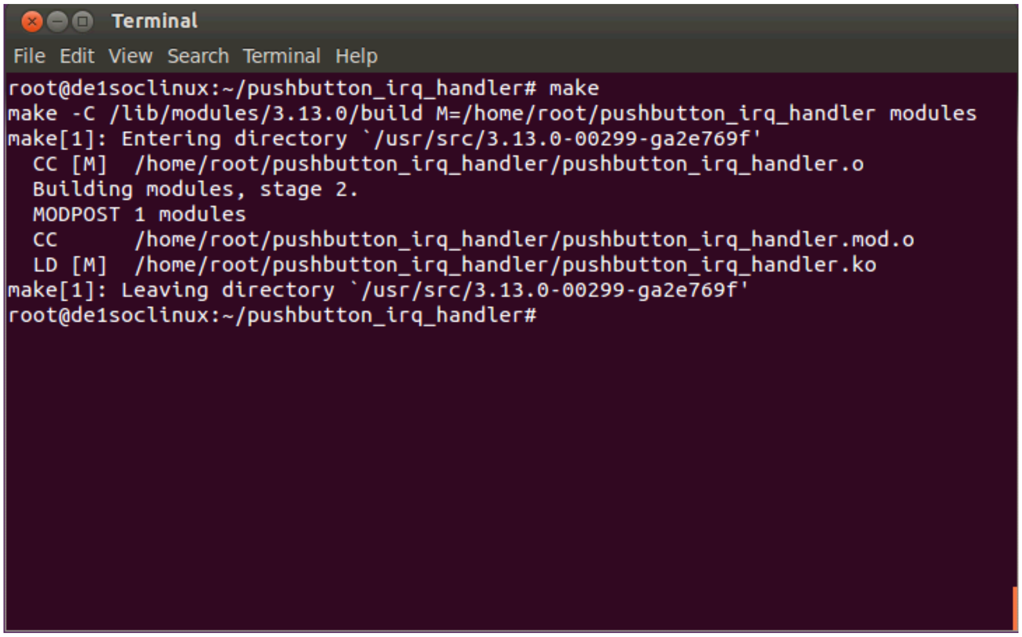

Figure 18. Inserting and removing the kernel module

Appendix A Using `Code::Blocks`

If you are using a network connection to your DE1-SoC Computer, as described in Section 2.7, then you can make use of the `Code::Blocks` tool for developing and debugging application programs. In this appendix we provide a simple example that shows how to create a `Code::Blocks` *project* for the DE1-SoC Computer. We will show how to build a project using the *increment_leds* example that we discussed in Section 3.3.

Open the `Code::Blocks` tool by clicking on its icon, which looks like four colored cubes. The main window of `Code::Blocks` is displayed in Figure 19. 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 20, 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.

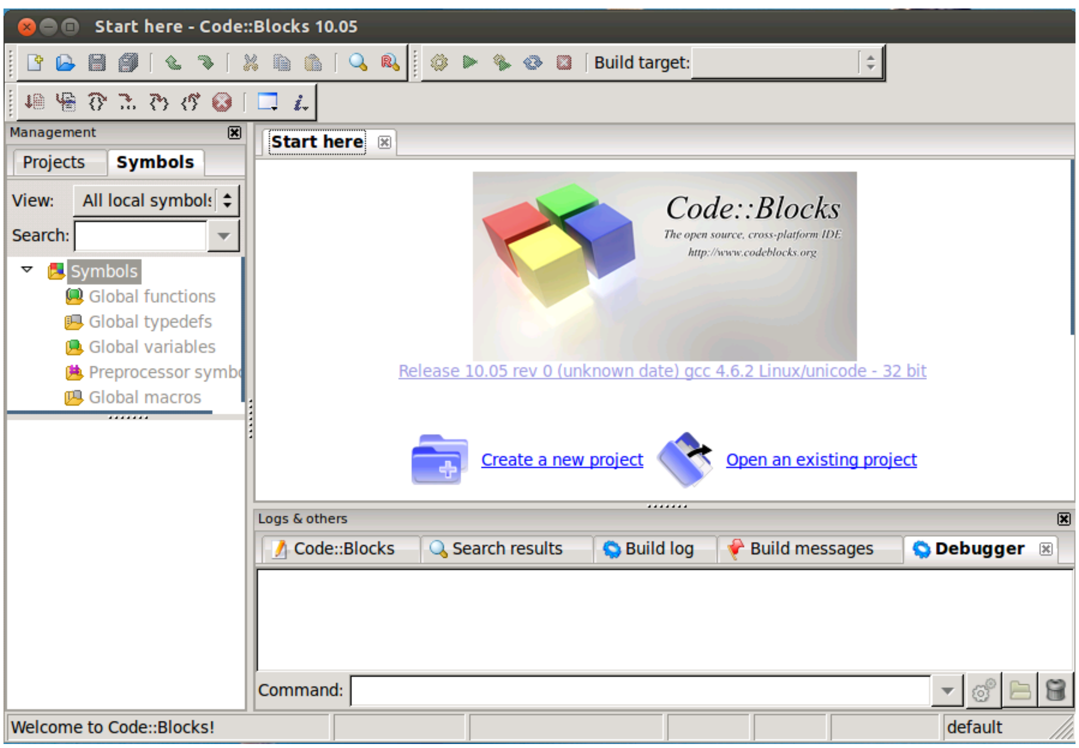

Figure 19. The main `Code::Blocks` window.

As indicated in Figure 21 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 22, and then click the `Open` button. In the dialog that opens, illustrated in Figure 23, 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.

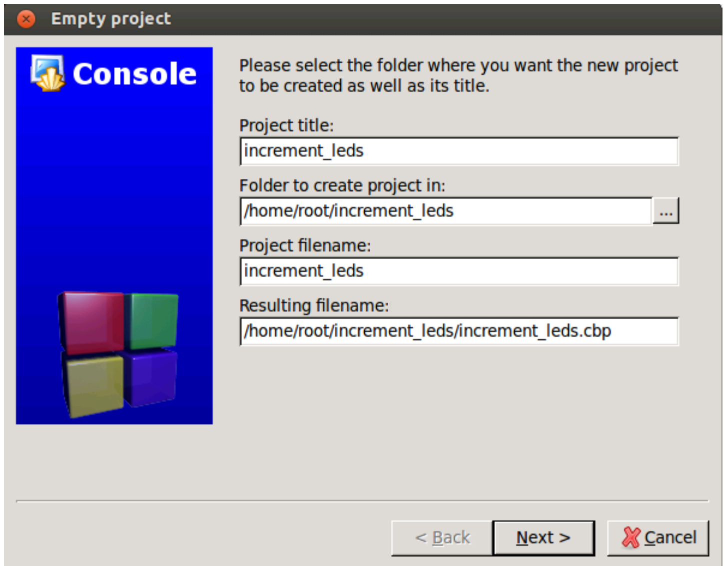

Figure 20. The `Empty project` dialog.

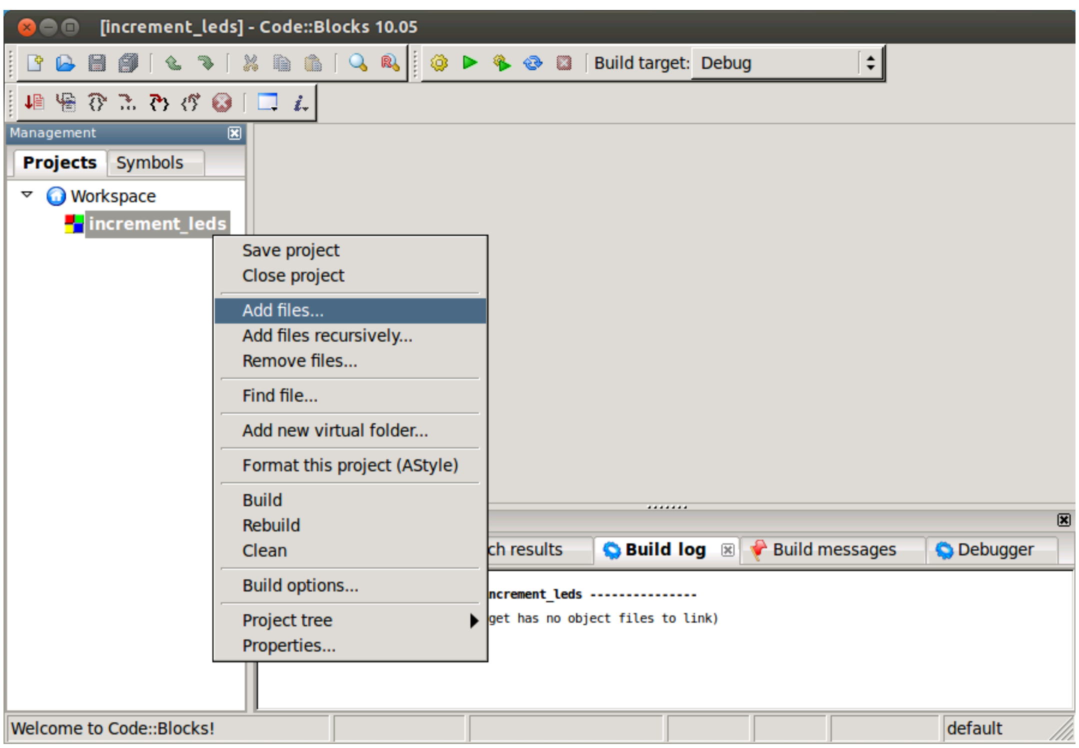

Figure 21. Adding source-code files to the project.

Click to the right of the line of code that calls the function `open_physical`, as shown in Figure 24, and set a *breakpoint*. The breakpoint is indicated by a small red circle.

Now start the debugger by selecting the command `Debug | Start`, as indicated in in Figure 25, (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 26 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 27. 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.

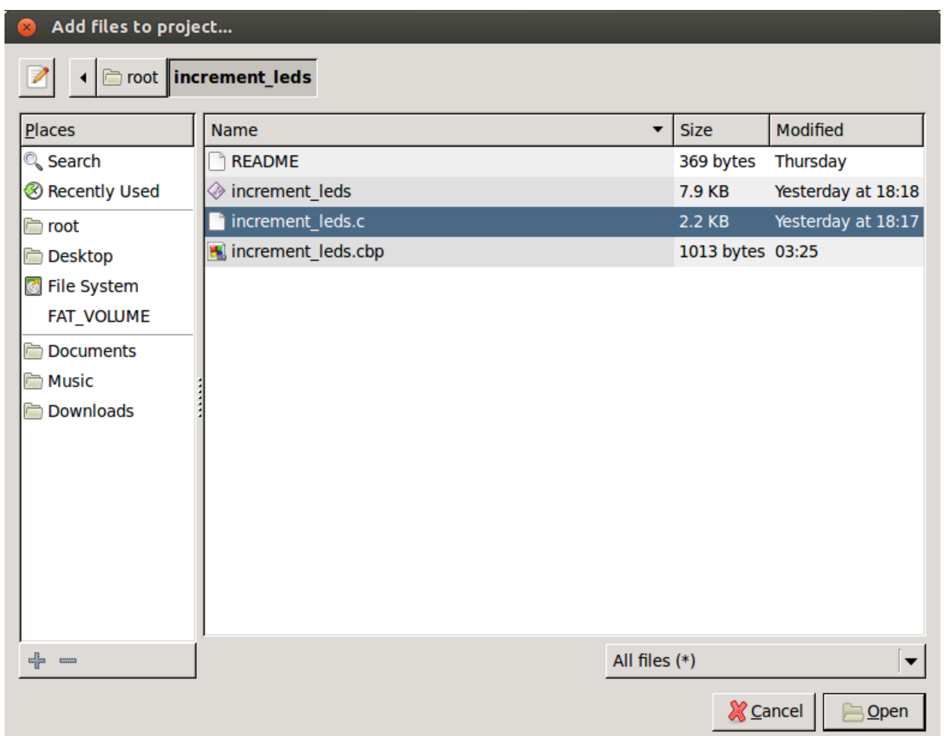

Figure 22. Selecting the *increment_leds.c* file.

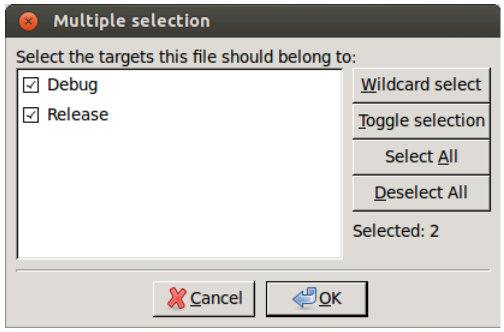

Figure 23. Selecting build targets.

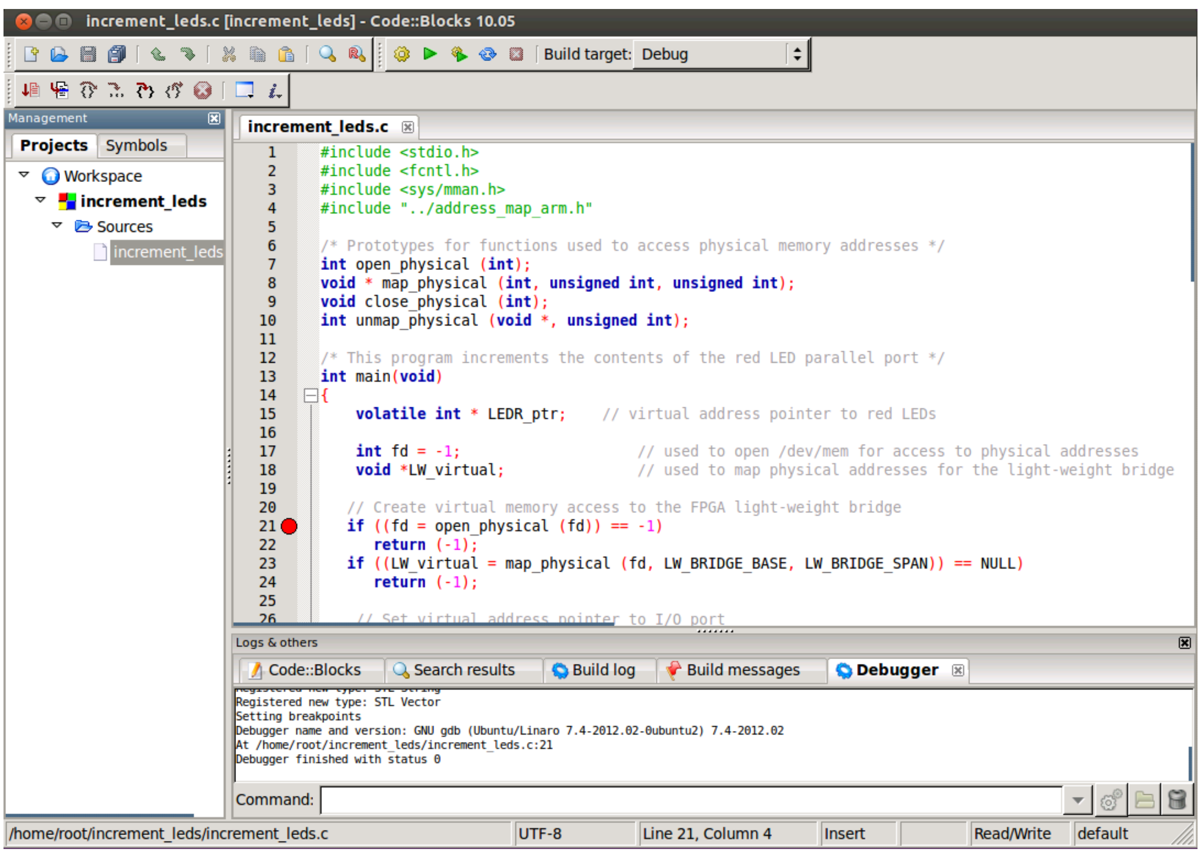

Figure 24. Setting a breakpoint.

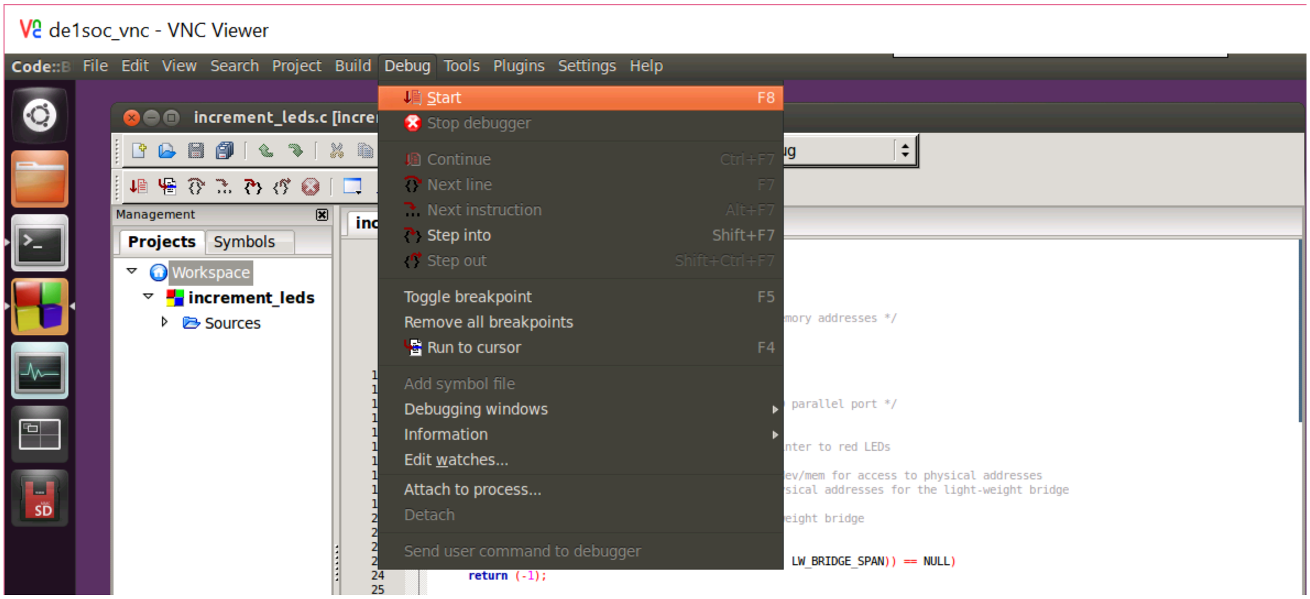

Figure 25. Starting the debugger.

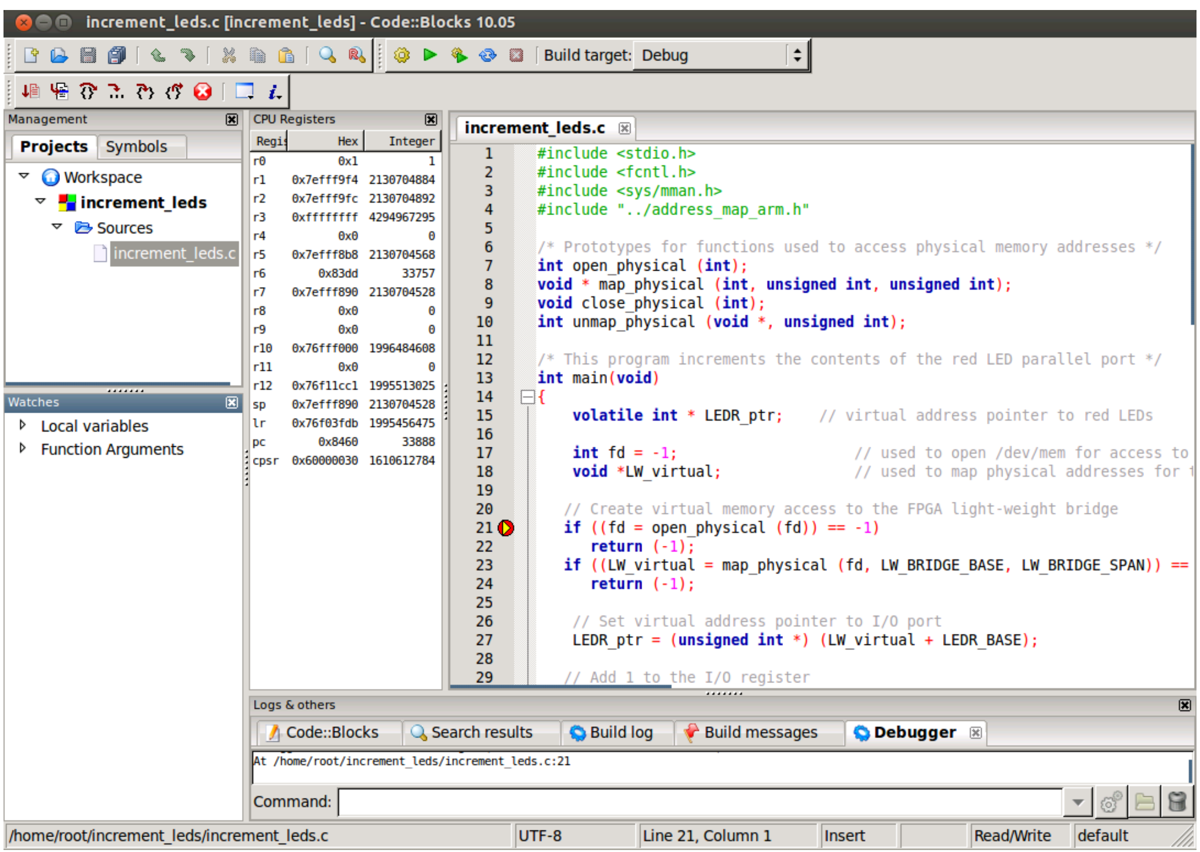

Figure 26. The debugging window.

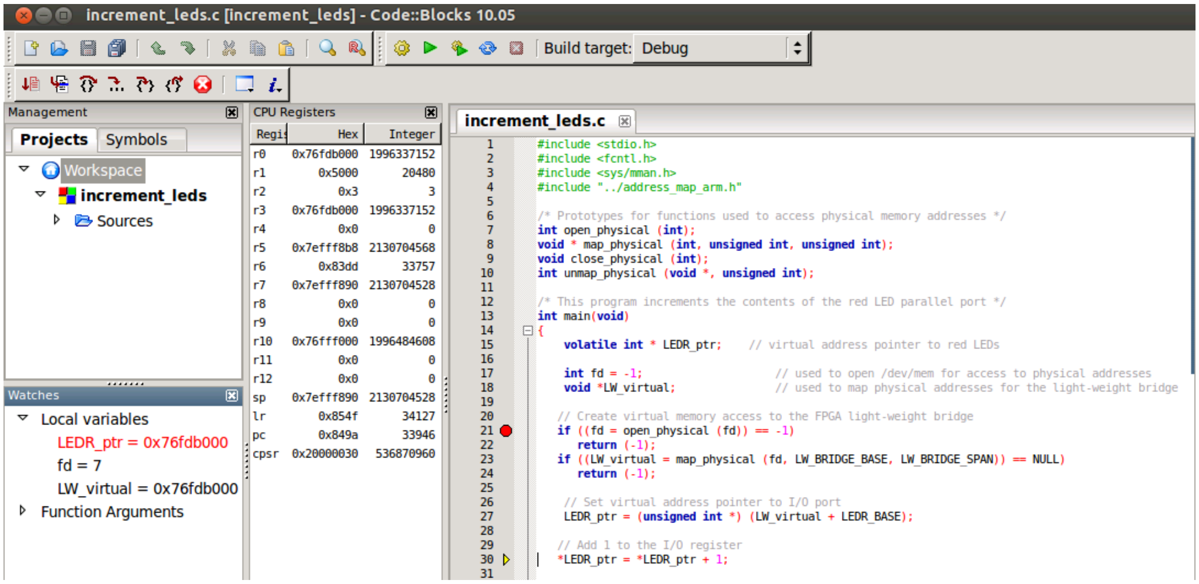

Figure 27. Displaying the values of variables.

Appendix B Include Files

Figure 28 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 TIMER_BASE            0x00002000
#define AV_CONFIG_BASE        0x00003000
#define PIXEL_BUF_CTRL_BASE   0x00003020
#define CHAR_BUF_CTRL_BASE    0x00003030
#define AUDIO_BASE            0x00003040
#define VIDEO_IN_BASE         0x00003060
#define ADC_BASE              0x00004000

#define LW_BRIDGE_SPAN        0x00005000
```

Figure 28. The contents of the file *address_map_arm.h*.

Figure 29 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   INTERVAL_TIMER_IRQ     72
#define   KEYS_IRQ               73
#define   FPGA_IRQ2              74
#define   FPGA_IRQ3              75
#define   FPGA_IRQ4              76
#define   FPGA_IRQ5              77
#define   AUDIO_IRQ              78
#define   PS2_IRQ                79
#define   JTAG_IRQ               80
#define   IrDA_IRQ               81
#define   FPGA_IRQ10             82
#define   JP1_IRQ                83
#define   JP2_IRQ                84
#define   FPGA_IRQ13             85
#define   FPGA_IRQ14             86
#define   FPGA_IRQ15             87
#define   FPGA_IRQ16             88
#define   PS2_DUAL_IRQ           89
#define   FPGA_IRQ18             90
#define   FPGA_IRQ19             91
```

Figure 29. The contents of the file *interrupt_ID.h*.

Appendix C Cross Compiling

When a program is compiled for an architecture that is different from that of the system doing the compiling, the process is called *cross-compilation*. In this section we will cross-compile a program for the ARM architecture, to run on the DE1-SoC Computer, from a host computer which typically runs on the x86 architecture. To do this, we will use a *gcc* toolchain that comes with the *Altera SoC EDS* suite. Specifically, we will use the `arm-linux-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 30. Save this code as `helloworld.c` in a folder of your choice on your host computer.

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");

    return 0;
}
```

Figure 30. 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 31. Navigate to the folder that contains the helloworld.c file by using the `cd` command.

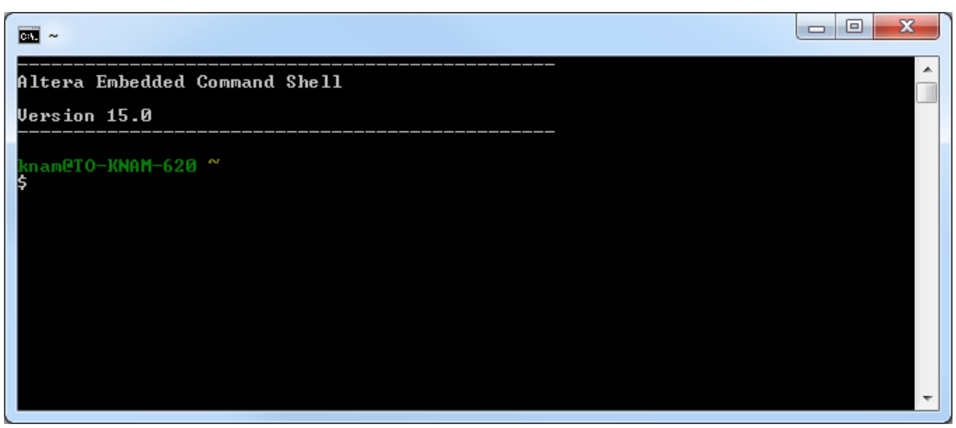

Figure 31. The Embedded Command Shell

Compile the code using the command `arm-linux-gnueabihf-gcc helloworld.c -o helloworld`, as shown in Figure 32. This command creates the output file *helloworld*, which is an ARM binary executable that we can copy to our Linux microSD card and execute on the DE1-SoC Computer. To copy the executable file you can either use an *ftp* program as discussed in Section 2.7.4, or you can follow the instructions in Section 3.4.3.

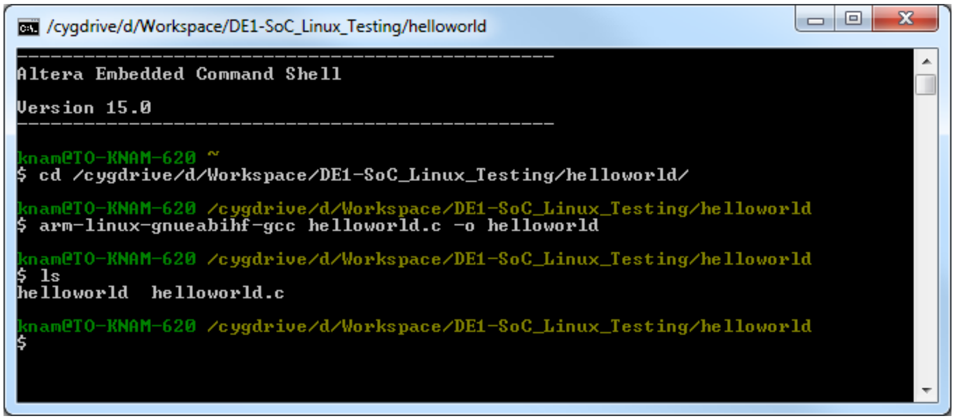

Figure 32. Cross-compiling the helloworld program

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

Transferring Files to the Linux Filesystem

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

From a Windows Host PC

When the microSD card is plugged into a Windows host PC, the FAT32 partition of the microSD card is detected. Any files that you move to this partition can be found in the */media/fat_partition/* directory of the Linux filesystem once Linux boots on the DE1-SoC board. Note that this partition will by default contain the files *soc_system.rbf* (an intermediate FPGA programming file used during boot up), *socfpga.dtb* (the device tree file), and *uImage* (the Linux kernel). Under no circumstances should you delete these files, as they are crucial components required to boot Linux.

From a Linux Host PC

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

Appendix D FPGA Configuration

A special mechanism built into the Cyclone V SoC allows software running on the ARM processor (such as the Linux OS) to program the FPGA. The *DE1-SoC-UP* Linux distribution contains drivers for this mechanism, allowing us to program the FPGA from the CLI. The following sections describe how to use this mechanism.

Creating an RBF Programming File

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

1. Launch the Convert Programming File tool by selecting `File > Convert Programming Files...`.
2. Select `Raw Binary File (.rbf)` as the `Programming file type`.
3. Select `Passive Parallel x16` as the `Mode`.
4. Specify the destination file name in the `File name` field.
5. Click and highlight `SOF Data` then add the .sof file that you wish to convert by clicking `Add File...`.
6. Click and highlight the newly added .sof file in the list, then select `Properties`. You should see the window shown in Figure 34. Enable file compression by ticking the checkbox as shown, then press OK.
7. We are now ready to generate the .rbf file. Click `Generate`. If all goes well, you will see the success message shown in Figure 35.
8. Finally, we can transfer the .rbf file to the Linux file system using the instructions provided in Section 3.4.3.

Programming the FPGA

The *DE1-SoC-UP* Linux distribution exposes the FPGA as the device file located at */dev/fpga0*. In addition, Linux provides files in */sys/class/fpga/* and */sys/class/fpga-bridges/* for probing the status and configuring settings of FPGA-related components. We will make use of these file-based interfaces to program the FPGA, using the steps described below.

1. Ensure that the MSEL switches on the DE1-SoC have been configured to MSEL[4:0] = 5'b01010.
2. Before we reprogram the FPGA with new components, we must first disable the FPGA-HPS bridges to avoid unpredictable behavior. Disable them using the following commands:
   - echo 0 > /sys/class/fpga-bridge/fpga2hps/enable
   - echo 0 > /sys/class/fpga-bridge/hps2fpga/enable
   - echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable

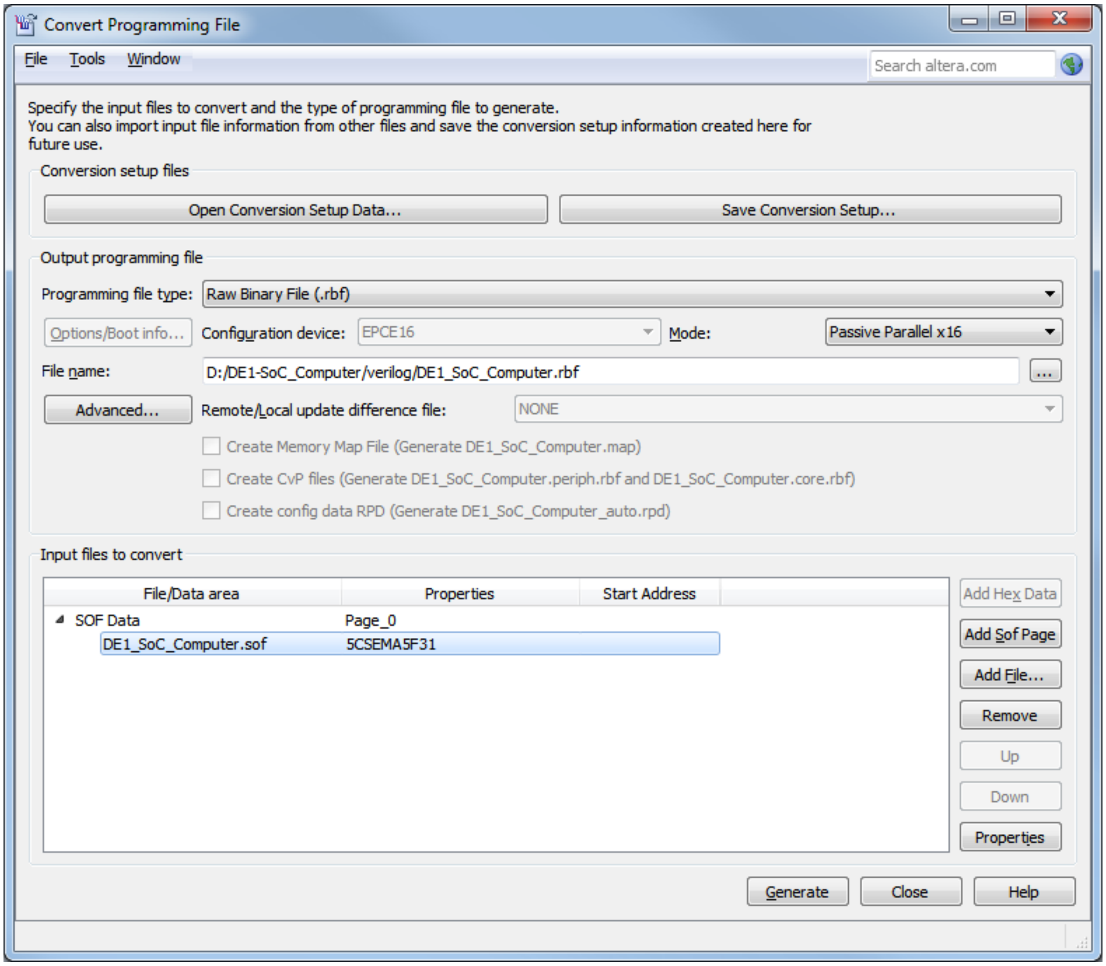

Figure 33. The Convert Programming File Tool.

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

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

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

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

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

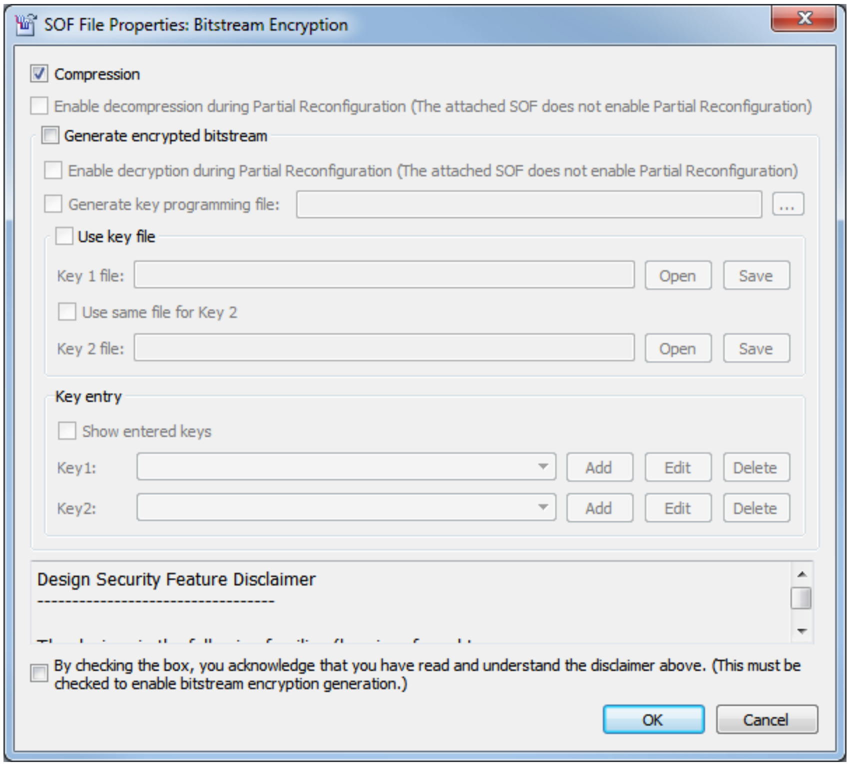

Figure 34. Enabling file compression.

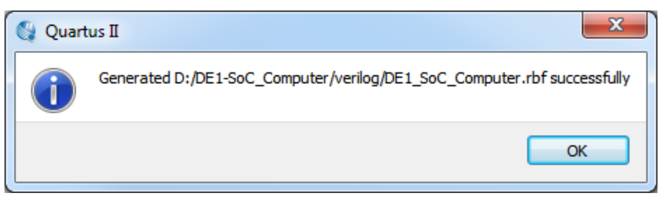

Figure 35. The .rbf file successfully generated.

Changing the Default FPGA Programming File

While the *DE1-SoC-UP* Linux distribution boots, scripts are executed to initialize various Linux components. One of these scripts, located at */etc/init.d/programfpga*, programs the FPGA with the default programming file */home/root/DE1_SoC_Computer.rbf*. If you open the script using a text editor such as `vi`, you will see that the script executes the FPGA programming commands described in Section 3.4.3. To change the default FPGA programming file, edit the line `dd if=<.rbf file> of=/dev/fpga0 bs=1M` to specify an .rbf file of your choosing.