Using Linux on the DE1-SoC

For Quartus II 15.0

1 Introduction

This tutorial describes the use of Linux with Altera SoC devices, with emphasis on using Linux with the Altera DE1-SoC development board containing the Cyclone V SoC device. It describes how to boot up Linux on the board, as well as how to use Altera SoC-specific Linux features such as the ability to program the FPGA from Linux commandline. Finally it describes how to write user-level and driver-level Linux programs that communicate with FPGA-side components.

Contents:

- Getting Started with Linux on the DE1-SoC board.
- Configuring the FPGA from Linux.
- Developing Linux Applications with FPGA Communication.
- Developing Linux Drivers for FPGA Components.

Requirements:

- Altera DE1-SoC Development Board.
- Micro-USB cable (for the UART-USB connector).
- MicroSD card (8GB or larger) and microSD card reader.

Optional (required for select optional sections):

- Altera Quartus II Software (required for Section 3.1).
- Altera SoC Embedded Design Suite (required for Section 4.2).

2 Running Linux on the DE1-SoC Board

Linux is an operating system found in a wide variety of computing devices such as personal computers, servers, and mobile smartphones. There are many reasons for using the Linux operating system - a list far too long to describe at length in this tutorial. A key advantage that we will leverage in this tutorial is Linux's built-in drivers that support a vast array of devices, including many of the devices found on the DE1-SoC board. Consider the USB and ethernet ports of the DE1-SoC board. Writing driver code for these devices is no easy feat, and would significantly increase the development time of an application that requires them. Instead, a developer can use Linux and its driver support for these devices, allowing them to use the high-level abstraction layers provided by the drivers to use the devices with minimal effort.

2.1 Preparing a Linux microSD Card

The DE1-SoC board is designed to boot Linux from an inserted microSD card. Altera and Terasic Technologies provide a number of Linux microSD card images that you can use to quickly get Linux running on the DE1-SoC. These Linux images range from a simple commandline-only Linux distribution, to the more full-featured Ubuntu Linux distribution with a GUI interface. To run one of these Linux distributions, you must write the image (provided in the .img file format) onto a sufficiently large microSD card. For this tutorial we will use the *DE1-SoC-UP-Linux.img* image, which accompanies this tutorial. To write the microSD card, we will use the free-to-use Win32 Disk Imager tool which you can find online using your favorite search engine. The steps involved in writing the microSD card are described below.

1. Plug in the microSD card to your computer using a microSD card reader, then launch Win32 Disk Imager.

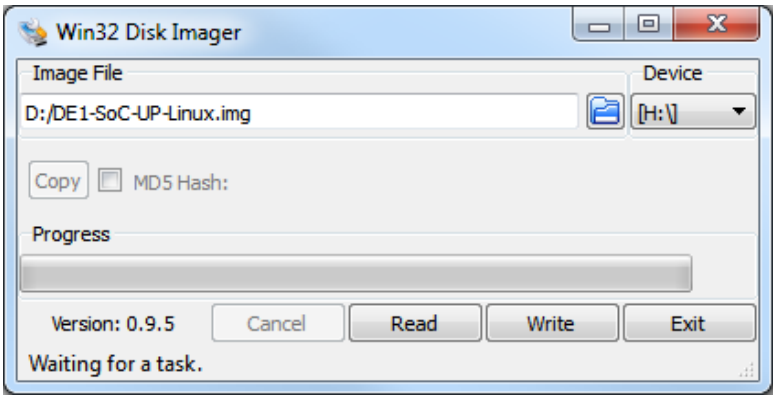

Figure 1. The Win32 Disk Imager tool

2. Select the drive letter corresponding to the microSD card under Device, as shown in Figure 1.
3. Select the DE1-SoC-UP-Linux.img image under Image File, as shown in Figure 1.
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.

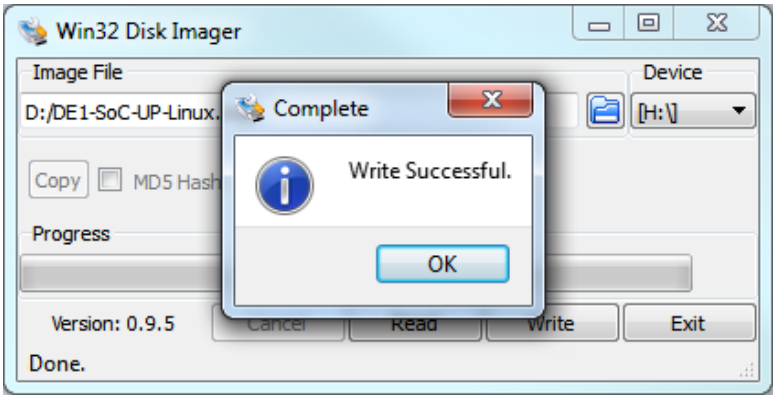

Figure 2. Win32 Disk Imager upon successfully writing the microSD card

2.2 Booting Linux on the DE1-SoC

Now that your microSD card is loaded with Linux, you can insert it into the microSD card slot on the DE1-SoC. Before turning on the board however, you must configure the MODE SELECT (MSEL) switches found on the underside of the board to MSEL[4:0] = 5'b01010, as shown in Figure 3. This configures the Cyclone V SoC chip to allow the ARM processor to program the FPGA, which is necessary as our Linux image will program the FPGA as part of its bootup process. In addition, we will need this MSEL configuration when we manually program the FPGA from the Linux commandline in Section 3.

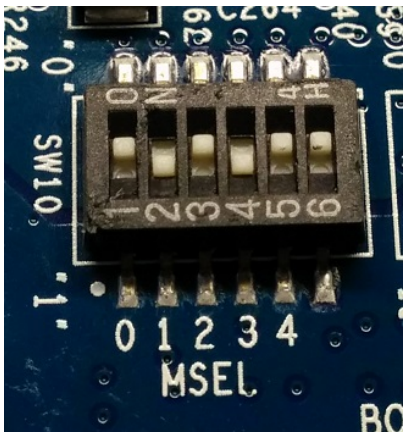

Figure 3. Configuring the MSEL switches of the DE1-SoC board

Once the microSD card is inserted and MSEL is configured, you can turn on the board to have Linux boot up. At this point, we require some way of interacting with the Linux OS. With a more full-featured Linux distribution with a GUI interface, you would at this point be able to connect a VGA monitor, a keyboard, and a mouse to interact with

the GUI displayed on the screen. However, the Linux distribution that we are using does not feature a GUI. Instead, it supports user interaction through the command line interface (CLI). In the following section, we will see how we can access the CLI from a host PC.

2.3 Accessing the Command Line Interface via UART Terminal

The Linux image we are using has been configured to route all of its text input and output (on the standard streams stdout, stdin, stderr) to the Cyclone V HPS's serial UART. The serial UART is a device that facilitates serial communication of characters, often through a serial cable. On the DE1-SoC, the serial UART is connected to a USB-to-UART chip which sends and receives the text through a USB cable to a host computer. On the host computer, we can use any of the readily available terminal programs that are capable of serial communication to access the CLI of the Linux running on the DE1-SoC.

For this tutorial, we will be using the free-to-use tool **Putty** which is available for both Windows and Linux. Start Putty, then connect the USB-to-UART of the DE1-SoC to your PC using a micro-USB cable. If this is your first time connecting to the USB-to-UART chip, you may have to install its device drivers on your host PC. The drivers can be downloaded at http://www.ftdichip.com/Drivers/VCP.htm.

On a Windows host PC, serial communication devices such as the USB-to-UART are recognized as COM ports. As there can be multiple COM ports connected to the PC, each COM port 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 PC. Here, there was only one COM port (the USB-to-UART) which was assigned the number 7 (COM7). In the case where there are many COM ports available, the number assigned to the USB-to-UART can be determined by disconnecting and reconnecting the cable to see which COM port disappears then reappears in the list.

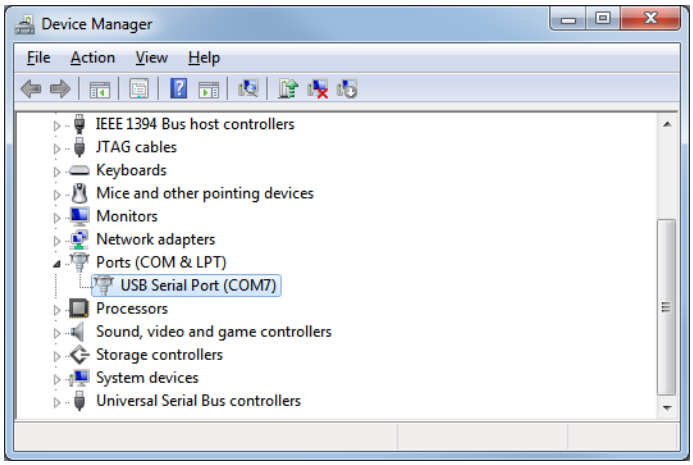

Figure 4. Determining the COM number assigned to the USB-to-UART in Device Manager.

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

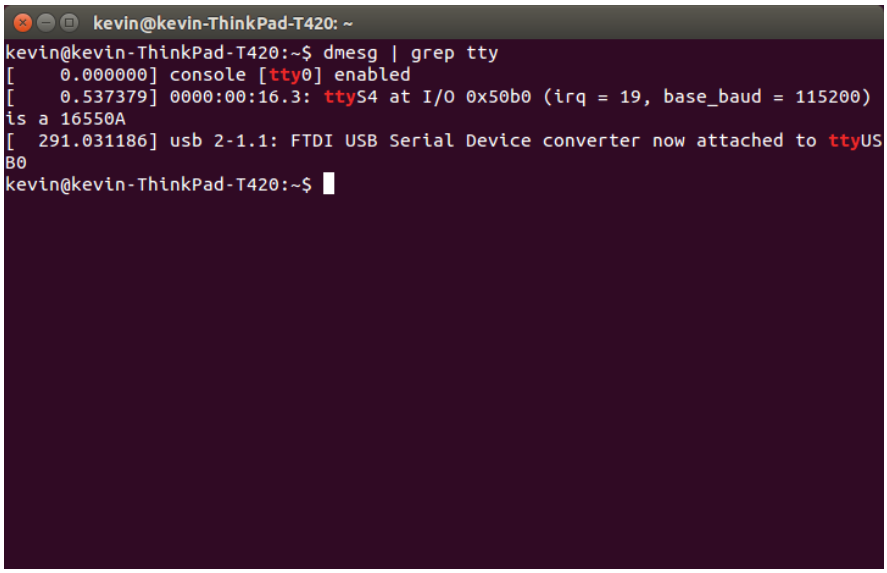

Figure 5. Determining the TTY device that corresponds to the USB-to-UART.

Once the serial device (COM port or TTY device) corresponding to the USB-to-UART is determined, Putty can be configured to connect to it. Figure 6 shows the main window of Putty. In this window, the Serial connection type must be chosen, and the COM port or TTY device must be entered in the Serial line field, as shown in the figure.

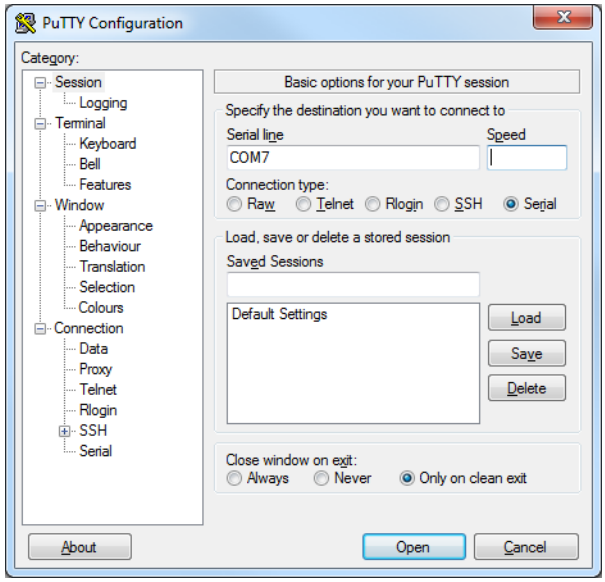

Figure 6. Putty's main window.

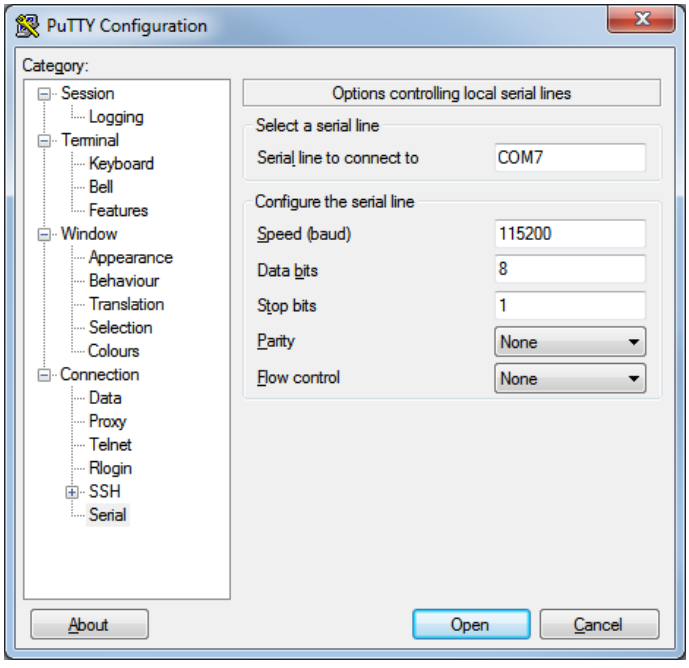

Figure 7. Putty's configuration window for serial communication settings.

Some additional details about the USB-to-UART 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 be configured to match the settings shown, with the speed (baud rate) set to 115200 bits per second, data bits set to 8 bits, stop bits set to 1 bit, and parity and flow control both set to none.

Once all of the serial line settings have been entered, press Open to start the terminal. You are now connected to the CLI and can start using Linux by entering Linux commands (try pressing Enter to draw the command prompt line). For illustrative purposes, press the WARM RST button on the DE1-SoC to restart Linux and see the Linux boot process from the beginning. You should now see a stream of textual information in the terminal as Linux boots up, as shown in Figure 8. Once Linux has finished booting, you will be logged in to the CLI as the "root" user, which means you have administrator-level privileges allowing you to modify settings and execute programs as you please.

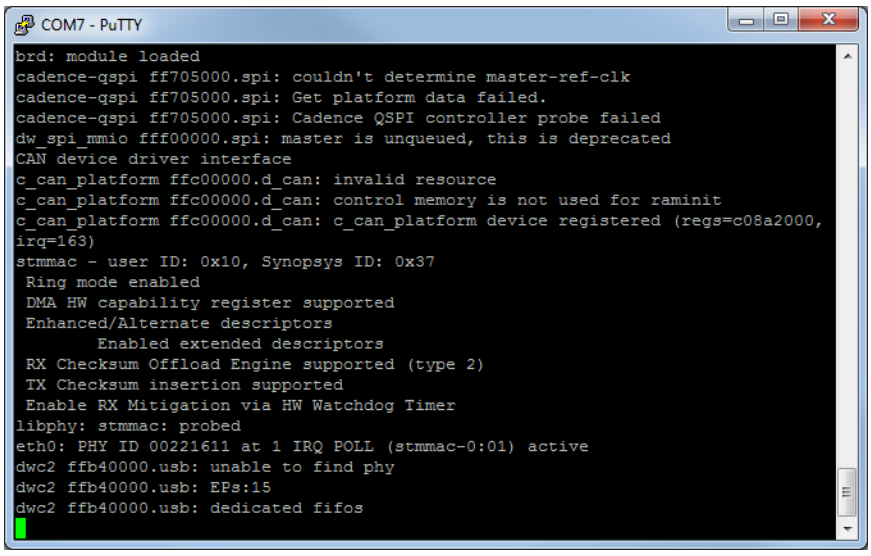

Figure 8. Putty terminal displaying text output as the Linux kernel boots up.

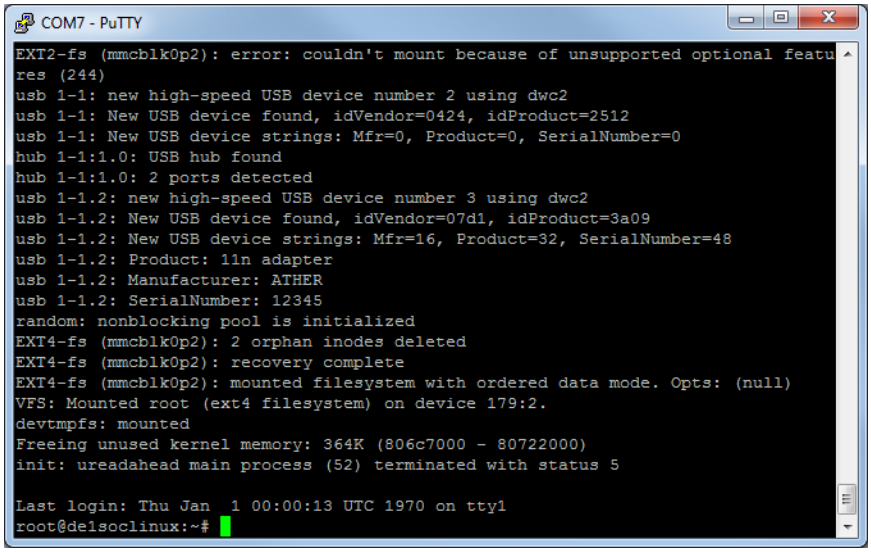

Figure 9. Logging in as root to the Linux command line interface.

2.4 Transferring Files to the Linux File System

2.4.1 From a Windows Host PC

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 microSD card. The Linux microSD card contains four partitions, but only one of them (the FAT32 partition) is recognized by a Windows host PC when the microSD card is plugged into the machine using a microSD card reader. Any files that you move to this partition can be found in the */media/fat_partition/* directory of the Linux filesystem once Linux boots. 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.

2.4.2 From a Linux Host PC

Transferring files to the Linux microSD card from a Linux host machine is no different than transferring files to a USB drive. Simply plug in the microSD card to the host machine via microSD reader, and copy over desired files. Note that there are two partitions to which you can transfer files. 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. 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.

3 FPGA Configuration Using Linux

A special mechanism built into the Cyclone V SoC allows software running on the ARM processor of the HPS to program the FPGA on the fly. Linux contains drivers for this mechanism, allowing users to program the FPGA with just a few commands through the CLI. This section describes how we can make use of this feature.

3.1 Creating an RBF Programming File (Optional)

The FPGA programming mechanism accepts the **Raw Binary File** (.rbf) file format as the input when programming the FPGA. For the purposes of this tutorial, sample .rbf files have been preloaded onto the Linux microSD card that you can use to try out the FPGA programming mechanism. If you would like to create your own .rbf file, you must convert a .sof file (the default programming file generated by Quartus) into an .rbf using Quartus's **Convert Programming File** tool. The steps for doing the file conversion 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....

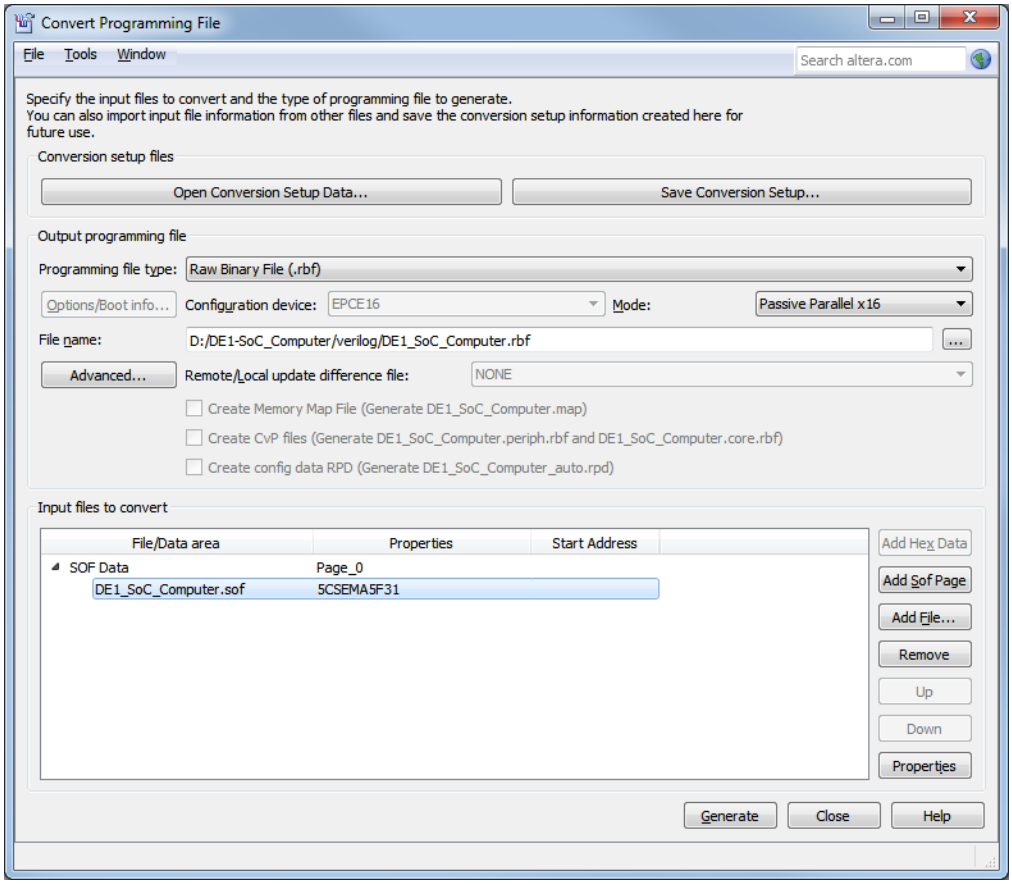

Figure 10. The Convert Programming File Tool.

6. Click and highlight the newly added .sof file in the list, then select Properties. You should see the window shown in Figure 11. Enable file compression by ticking the checkbox as shown, then press OK.

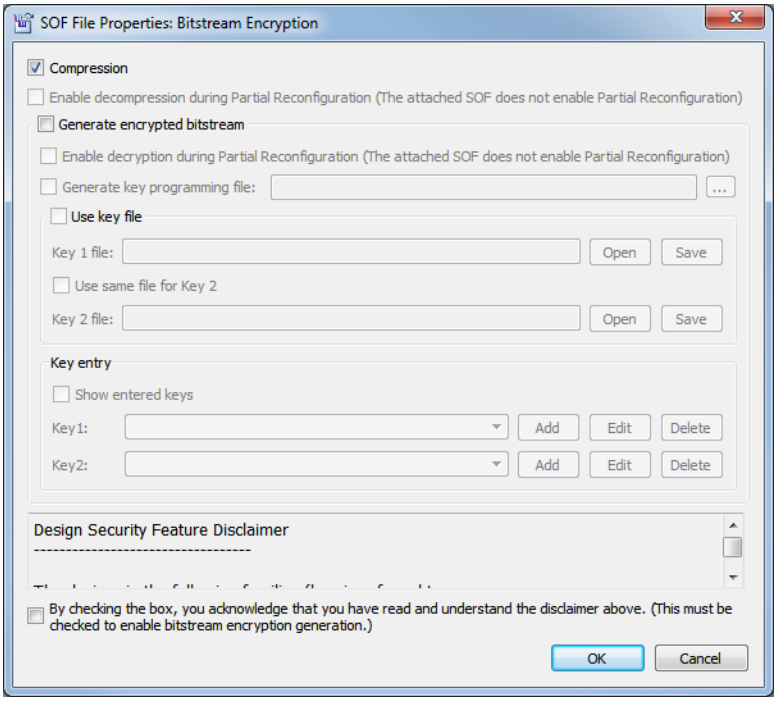

Figure 11. Enabling file compression.

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 12.

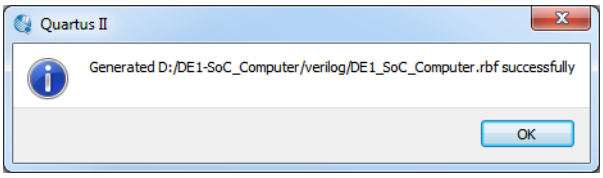

Figure 12. The .rbf file successfully generated.

8. Finally, we can transfer the .rbf file to the Linux file system using the instructions provided in Section 2.4.

3.2 Programming the FPGA

In Section 4, we will be creating and running programs that communicate with the FPGA. These programs will require that the FPGA has been programmed with the DE1-SoC Computer system, located at */home/root/DE1_SoC_Computer.rbf*. Luckily for us, our Linux image automatically programs the FPGA with this system while it is

booting up, so we do not manually have to do so. In the future you may wish to program the FPGA with your own .rbf file. This section describes the steps required to program the FPGA from the Linux CLI.

The Linux OS exposes the devices present in the system as files in the /dev/ directory. The FPGA device is represented by the file /dev/fpga0. In addition, Linux provides files in the /sys/class/ directory that are meant for probing and configuring various devices. In the case of the FPGA, Linux provides files in the /sys/class/fpga/ directory as well as the /sys/class/fpga-bridges/ directory that let us check the status of the FPGA-related components, and configure various settings. 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. Disable the FPGA-HPS bridges (hps2fpga, fpgs2hps, and lwhps2fpga) 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

   Explanation: the FPGA-HPS bridges facilitate communication between the HPS and FPGA-side components. Since we are about to (re)program the FPGA with new components, we must first disable these bridges to avoid unpredictable behavior.
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

3.3 Changing the Default FPGA Programming File (Optional)

As part of the Linux boot up, various scripts are executed to initialize various Linux components. In the Linux image that we are using, a script located at */etc/init.d/programfpga* is executed as part of the startup, which 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 3.2. If you would like to change the default FPGA programming file, simply edit the line `dd if=<.rbf file> of=/dev/fpga0 bs=1M` to specify an .rbf file of your choosing.

4 Developing Linux Programs for DE1-SoC

In this section, we will explore developing programs for Linux on the DE1-SoC. As we will see, writing and compiling programs for this system is very similar to writing and compiling any Linux program; after all, Linux is Linux whether it runs on the ARM of the DE1-SoC or an Intel CPU in your workstation.

There are two options for compiling and running a Linux program for the DE1-SoC. The first is to write and compile code within the commandline interface of the Linux running the board. This approach is described in Section 4.1. The second is to write and compile the code from a host computer, then copy over the resulting executable to the Linux microSD card. This approach is described in Section 4.2.

4.1 Native Compilation on the DE1-SoC

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 commandline interface, using its built-in compilation toolchain.

To demonstrate native compilation, we will compile a simple helloworld program. The code for this program is shown below in Figure 13. You can also find the code in */home/root/helloworld/helloworld.c*.

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

Figure 13. The helloworld program

Change the present working directory to the directory that contains the source code using the command cd /home/-root/helloworld. Compile the program using the command gcc helloworld.c -o helloworld, as shown in Figure 14. gcc stands for 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 the code that we wish to compile. The second is -o helloworld which tells the compiler to produce an output executable named "helloworld". Once the compilation is complete, you can run the program by typing ./helloworld. The program will output the message "Hello World!" then exit, as shown in Figure 14.

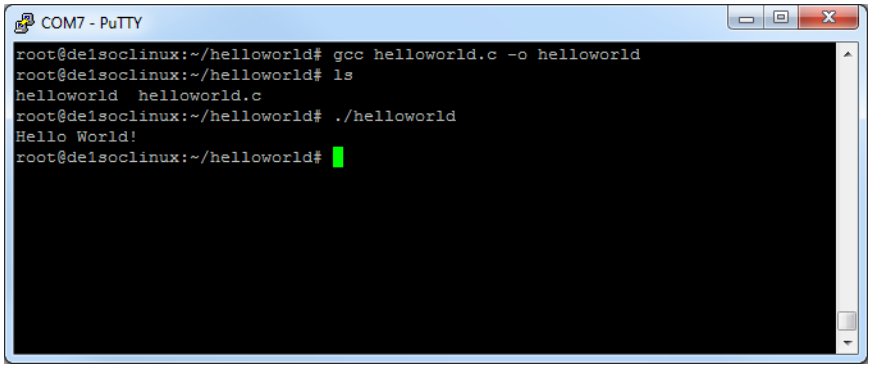

Figure 14. Compiling and executing the helloworld program through commandline

4.2 Cross Compilation from an x86 Host Computer (Optional)

When a program is compiled for an architecture that is different than 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) from your host computer which is likely based 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 SoC EDS installation directory.

We will compile a simple helloworld program, the code for which is shown below in Figure 15. Save this code as helloworld.c in a folder of your choice.

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

Figure 15. 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 16. Navigate to the folder that contains the helloworld.c file by using the cd command.

Figure 16. The Embedded Command Shell

Compile the helloworld program using the following command: arm-linux-gnueabihf-gcc helloworld.c -o helloworld, as shown in Figure 17. This creates the output file *helloworld*, which is the helloworld ARM binary executable that we can copy to our Linux microSD card and execute on the DE1-SoC.

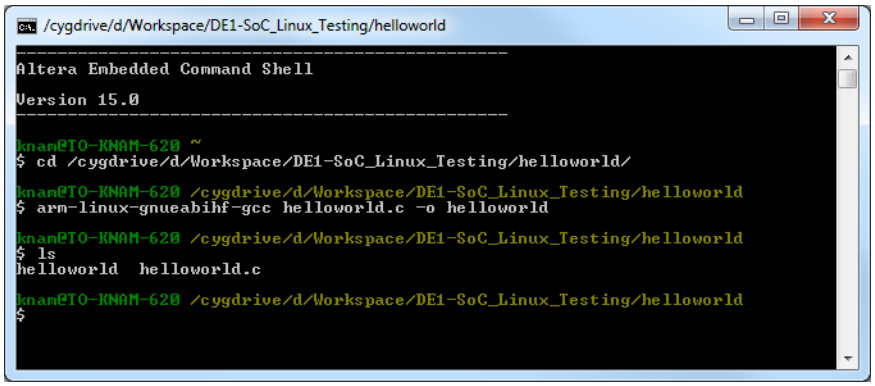

Figure 17. Cross-compiling the helloworld program

It should strike anyone familiar with the gcc toolchain that using the arm-linux-gnueabihf- toolchain is not much different from any other gcc toolchain. In addition to gcc, the arm-linux-gnueabihf- toolchain contains the familiar suite of gnu compilation programs such as the C++ compiler (g++), linker (ld), the assembler (as), and the object dump (objdump) and object copy (objcopy) utilities, all of whose executable names are prefixed by "arm-linux-gnueabihf-".

4.3 Communicating with the FPGA from a Linux Program

Programs running on the ARM processor of the Cyclone V SoC can communicate with FPGA-side components through the either the HPS-to-FPGA (hps2fpga) or the Lightweight HPS-to-FPGA (lwhps2fpga) bridge built into the chip. These bridges are mapped to regions in the ARM memory space, meaning that communicating with the FPGA effectively boils down to accessing these memory regions. When an FPGA-side component (such as an IP core) is connected to one of these bridges, the component's memory-mapped register interface will be available to the ARM within the bridge's memory region. Qsys is used to connect the component to a bridge, and to set its register interface's address offset within the bridge's memory span.

If we were developing a baremetal ARM program (a program that does not run on top of an operating system), accessing addresses within the bridge memory region would be as simple as dereferencing a pointer to the physical address of interest. For a program running on Linux, things are slightly complicated as Linux programs have access to a virtual memory space rather than the physical memory space.

In order to access physical memory addresses from a Linux program, we must use a function provided by the Linux kernel called `mmap`, together with the system memory device located at */dev/mem*. The `mmap` function, which stands for memory map, is a function that 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 memory addresses in the virtual memory span to which the file has been mapped. The system memory device file, */dev/mem*, is a file that represents the physical system memory. 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 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 lwhpw2fpga bridge's memory span and communicate with an IP core on the FPGA.

4.4 Example Program with FPGA Communication

Below is an example program that communicates through the lwhps2fpga bridge to alter the state of the red LEDs on the board. The program requires that the FPGA has been programmed with the DE1-SoC Computer system. If you have altered the FPGA programming, you must program the FPGA manually with the file located at */home/root/DE1-SoC_Computer.rbf*. The program makes use of the Parallel I/O (PIO) IP core in the DE1-SoC Computer that is connected to the LEDs. The core provides a memory-mapped register that allows a master such as an ARM processor to write a desired value to the LEDs. Critical for the purposes of our program is that the register interface has been connected to the lwhps2fpga bridge, allowing the ARM processor to access the memory-mapped register.

When the DE1-SoC Computer was configured in Qsys, the PIO core was given the address offset 0x0 within the address span of the lwhps2fpga bridge. The end result is that PIO's register is mapped to address 0xff200000 (base address of the lwhps2fpga bridge 0xff200000 + offset 0x0 = 0xff200000) in ARM memory space. As we will see, the program alters the LEDs by simply writing to the physical address 0xff200000.

The high-level behavior of the program is as follows. First, it reads the current value in the PIO register. Then, it adds one to the value and writes the incremented value back into the register. As you will see when you execute this program, the value being displayed on the LEDs will increment by one each time the program is run (try running it multiple times). Important lines of the code are described in further detail below:

- Line 4: the `sys/mman.h` header file is included, which provides the mmap and munmap functions required for this program.
- Line 18 opens the file */dev/mem* which represents the system's physical memory. We will map this file into memory using `mmap`, which will allow us to write to physical memory addresses.
- Line 24 maps a part of the */dev/mem* file into memory. Specifically, it maps a portion of the file `HW_REGS_SPAN` wide, starting at address `HW_REGS_BASE` into memory. This window covers the entire lwhps2fpga memory span, allowing us to access any FPGA-side registers that have been connected to the bridge. The `mmap` function returns the virtual address that maps to the bottom of the physical memory stretch that we requested (`HW_REGS_BASE`). This means an access to `virtual_base + offset` will access the physical address `0xff200000 + offset`.
- Line 32 calculates the virtual address that maps to the LED PIO register. This is done by adding the address offset of the PIO register `LED_PIO_BASE` (the offset into the bridge span) to `virtual_base`.
- Line 35 reads the LED PIO register, increments the value by one, then writes the incremented value back to the register.
- Line 37 unmaps the /dev/mem file from memory, now that we have finished accessing physical memory.

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

#define HW_REGS_BASE ( 0xff200000 )
#define HW_REGS_SPAN ( 0x00200000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )
#define LED_PIO_BASE 0x0

int main(void)
{
    volatile unsigned int *h2p_lw_led_addr=NULL;
    void *virtual_base;
    int fd;

    // Open /dev/mem
    if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
        printf( "ERROR: could not open \"/dev/mem\"...\n" );
        return( 1 );
    }

    // get virtual addr that maps to physical
    virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ),
       MAP_SHARED, fd, HW_REGS_BASE );
    if( virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap() failed...\n" );
        close( fd );
        return(1);
    }

    // Get the address that maps to the LEDs
    h2p_lw_led_addr=(unsigned int *)(virtual_base + (( LED_PIO_BASE ) & (
       HW_REGS_MASK ) ));

    // Add 1 to the PIO register
    *h2p_lw_led_addr = *h2p_lw_led_addr + 1;

    if( munmap( virtual_base, HW_REGS_SPAN ) != 0 ) {
        printf( "ERROR: munmap() failed...\n" );
        close( fd );
        return( 1 );

    }
    close( fd );
    return 0;
}
```

Figure 18. C-code for the increment_leds program

4.5 Handling Interrupts

In this section we will explore how to implement an interrupt handler in Linux. Specifically, we will handle interrupts generated by the four push buttons on the DE1-SoC. To accomplish this, we will create a **kernel module**, which is a special program that can be "inserted" into the Linux kernel. Once inserted, a kernel module executes as part of the kernel with many privileges that are not available to user-level programs, crucially among which is the ability to register an interrupt handler.

The ARM processor of the Cyclone V HPS contains 256 interrupt request (IRQ) lines ranging from IRQ0 to IRQ255. These IRQ lines are connected to various interrupt-generating devices such as USB, ethernet, timers, etc. 64 of the lines (IRQ72 - IRQ135) are reserved for interrupts originating from FPGA side. These interrupt lines can be connected to any interrupt-generating components when building a system in Qsys. In the DE1-SoC Computer system, the interrupt sender of the pushbutton PIO core has been connected to IRQ73. This means that our kernel module needs to register an interrupt handler that will listen to IRQ73.

Linux makes it easy to register an interrupt handler. Since Linux contains drivers that handle the low-level intricacies of the Cyclone V HPS's interrupt handling mechanism, we can use a high-level API provided by Linux to register our handler. In our kernel module code, we can simply include the `linux/interrupt.h` header file which provides the function `request_irq(unsigned_int irq, irq_handler_t handler, unsigned long irqflags, const char* devname, void * dev_id)`. The important arguments to this function for our purposes is `irq`, which for us is 73, and `handler` which is a function pointer to our custom interrupt handling function.

4.5.1 The Pushbutton Interrupt Handler Kernel Module

The code for our kernel module is shown in Figure 19. You may notice that unlike a user-level program, 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 with the macros `module_init(...)` and `module_exit(...)`.

The init function in our module is `initialize_pushbutton_handler(void)`. This function sets the LEDs to 0x200, which turns on the leftmost LED to indicate that the module has been inserted. Then it configures the pushbutton PIO core to start generating interrupts on button presses. Finally, it calls `request_irq(...)` to register our irq handler `irq_handler(...)` to handle pushbutton interrupts.

Once registered, `irq_handler(...)` is executed whenever the pushbutton PIO core generates an interrupt. The handler does two things. First, it increments the value displayed on the LEDs to indicate that the interrupt has been handled. Second, it clears the interrupt in the PIO core by clearing the PIO core's edgecapture register to allow it to generate a new interrupt on the next button press.

The exit function in our module is `cleanup_pushbutton_handler(void)`, which 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
7  MODULE_LICENSE("GPL");
8  MODULE_AUTHOR("Altera University Program");
9  MODULE_DESCRIPTION("DE1SoC Pushbutton Interrupt Handler");
10
11 void * lwbridgebase;
12
13 irq_handler_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
14 {
15     // Increment the value on the LEDs
16     iowrite32(ioread32(lwbridgebase)+1,lwbridgebase);
17     // Clear the edgecapture register (clears current interrupt)
18     iowrite32(0xf,lwbridgebase+0x5c);
19
20     return (irq_handler_t) IRQ_HANDLED;
21 }
22
23 static int __init intitialize_pushbutton_handler(void)
24 {
25     // get the virtual addr that maps to 0xff200000
26     lwbridgebase = ioremap_nocache(0xff200000, 0x200000);
27     // Set LEDs to 0x200 (the leftmost LED will turn on)
28     iowrite32(0x200,lwbridgebase);
29     // Clear the PIO edgecapture register (clear any pending interrupt)
30     iowrite32(0xf,lwbridgebase+0x5c);
31     // Enable IRQ generation for the 4 buttons
32     iowrite32(0xf,lwbridgebase+0x58);
33
34     // Register the interrupt handler.
35     return request_irq(73, (irq_handler_t)irq_handler, IRQF_SHARED,
           "pushbutton_irq_handler", (void *)(irq_handler));
36 }
37
38 static void __exit cleanup_pushbutton_handler(void)
39 {
40     // Turn off LEDs and de-register irq handler
41     iowrite32(0x0,lwbridgebase);
42     free_irq(73, (void*) irq_handler);
43 }
44
45 module_init(intitialize_pushbutton_handler);
46 module_exit(cleanup_pushbutton_handler);
```

Figure 19. C-code for the pushbutton interrupt handler kernel module

4.5.2 Compiling the Kernel Module

The kernel module source code can be found in the directory */home/root/pushbutton_irq_handler/*. To compile the module, you can use the included Makefile by running the command make. The contents of the Makefile are shown in Figure 20. Let us take a closer look at the contents of the Makefile. First, the line `obj-m += pushbutton_irq_handler.o` specifies that the object should be compiled as a kernel module (denoted by the "-m"). This line also specifies the name of the kernel module that is generated (our kernel module will as a result be named pushbutton_irq_handler).

Next we see the "all" target (the default target when make is called), which calls the command `make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules`. The `-C /lib/modules/$(shell uname -r)/build` argument first changes the working directory to the build directory of the active Linux kernel. This directory contains the source code and the configuration files with which the currently running kernel was built. Once in the directory, the make command leverages the **kbuild** build system that exists within the directory, to build our kernel module. The kbuild build system is essentially a collection of special "Kbuild" makefiles that allows users to configure and compile the Linux kernel and related objects such as kernel modules.

Since our kernel module is external to the kernel source code, it is referred to as an **external kernel module**. As such, we must tell the kbuild system to build an external module with the argument `M=$(PWD)`, which provides it with the directory containing our kernel module source code. Finally, the `modules` target tells kbuild that we want to build the kernel modules located in the directory pointed to by the `M=` argument. The end result of the "all" target is the *pushbutton_irq_handler.ko* kernel module.

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

Figure 20. Kernel module makefile

4.5.3 Running the Kernel Module

Once compiled, the kernel module is executed by "inserting" it into the Linux kernel. To insert a module, you can use the command insmod <path to .ko file>. Let's insert our module using the command insmod pushbutton_irq_handler.ko, as shown in Figure 21. Using the command lsmod, which lists the currently loaded kernel modules, you can confirm that our module "pushbutton_irq_handler" is in the list. On the DE1-SoC board, you should see that the leftmost LED is turned on, indicating that the module has been loaded. Try pressing any of the four push buttons to generate an interrupt on IRQ73, and confirm that the IRQ handler is working as expected (the LEDs should increment).

To stop a kernel module, you must remove it from the kernel by using the command rmmod <kernel module name>. The module name is specified in the Makefile via the line `obj-m += <kernel module name>.o`. In our case, you can remove the module using the command rmmod pushbutton_irq_handler. Similar to before,

you can use the lsmod command to confirm that the pushbutton_irq_handler module is no longer in the list.

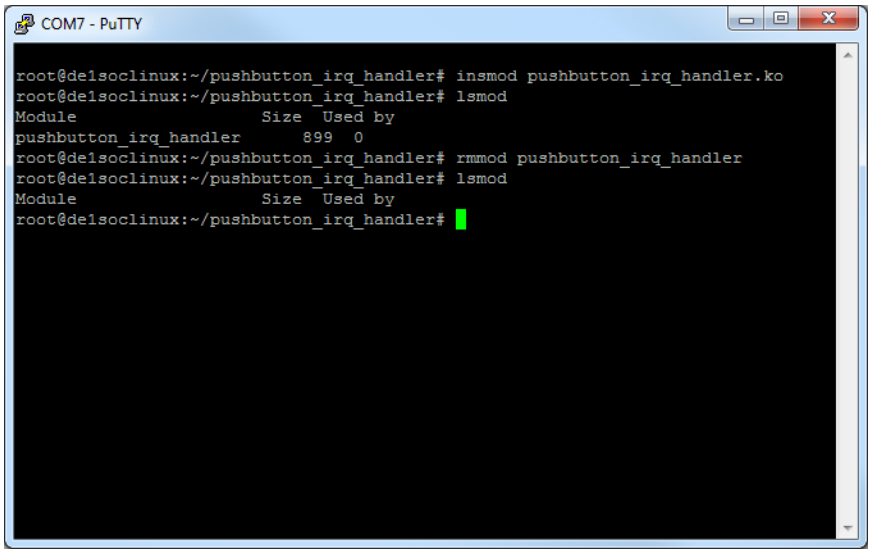

Figure 21. Inserting and removing the kernel module