Using PCI Express on DE4 Boards

1 Introduction

This tutorial describes how to use the PCI Express (Peripheral Component Interconnect) on Intel DE4 board. It first demonstrates how to build a system with the PCI Express IP Core using Platform Designer and then shows how to use the PCI driver on Linux* operating system. The discussion is based on the assumption that the reader has the basic knowledge of C language and Verilog hardware description language. Also the reader should be familiar with Quartus® Prime software and Linux operating system.

Contents:

- Background
- Building the PCI Express System
- Using the Driver
- Changing the Driver Codes
2 Background

PCI Express is a high-performance interconnect protocol for use in a variety of applications including network adapters, storage area networks, embedded controllers, graphic accelerator boards, and audio-video products.

An example of a system using PCI Express on DE4 board is shown in Figure 1. The PCI Express IP Core implements the PCI Express interface protocol to allow data transfer between a host computer and the on-chip memory on the board. The data can be transferred either directly or through the DMA controllers. Besides, a Nios® II processor is used to perform calculation on the data in the on-chip memory.

A simple application of this system is that the host computer can put data to the on-chip memory through a DMA controller and wait for the Nios II processor to perform calculation on the data. Once the data is calculated, the host computer can read the data back through the PCI Express IP Core.

Note that this tutorial shows the ways to use both DMA Controller and Scatter-Gather DMA Controller. In a real design, two different kinds of DMA controllers are not necessary.
3 Building the PCI Express System

To use the PCI Express on the DE4 board, you have to instantiate the PCI Express IP Core in your hardware system. To start with, create a new Quartus Prime project for your system. As shown in Figure 2, store your project in a directory called `de4_pcie_tutorial` and assign the same name to both the project and its top-level design entity. In your project, from the list of available devices, choose the appropriate device name for the FPGA used on the DE4 board. A list of devices names of DE4 boards can be found in Table 1.

![New Project Wizard](image)

Figure 2. Create a new project.

<table>
<thead>
<tr>
<th>Board</th>
<th>Device Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE4-230</td>
<td>Stratix® IV GX EP4SGX230KF40C2</td>
</tr>
<tr>
<td>DE4-530</td>
<td>Stratix® IV GX EP4SGX530KH40C2</td>
</tr>
</tbody>
</table>
3.1 Creating a Platform Designer System

In this section, we build a system with the PCI Express IP Core, DMA Controller, Scatter-Gather DMA Controller, on-chip memory, and the Nios II processor in Platform Designer. Select Tools > Platform Designer to open the Platform Designer as shown in Figure 3, and then save the file as qsys_system.qsys.

![Figure 3. Open the Platform Designer tool.](image)

To add the PCI Express IP Core, perform the following:

- Select Interface Protocols > PCI Express > IP Compiler for PCI Express and click Add. The Configuration Wizard window appears. You can use the scroll bar on the right to view parameters that are not initially visible.

- Under the System Settings heading, set the Test out width to None and leave others as default.

- Under the PCI Base Address Registers heading, set BAR0 as 64 bit Prefetchable Memory and BAR2 as 32 bit Non-Prefetchable.
• Under the **Device Identification Registers** heading, set the **Device ID** to 0x00000de4.

• Under the **Address Translation** heading, set the **Address translation table configuration** to *Dynamic translation table*, **Number of address pages** to 2 and the **Size of address pages** to 1 GByte - 30 bits.

• Click **Finish** to add the PCI Express IP Core `pcie_hard_ip_0` to the Platform Designer system. Figure 4 shows screenshots of the settings.

![IP Compiler for PCI Express](image1)

![IP Compiler for PCI Express](image2)

**Figure 4. Settings for PCI Express IP Core.**

To add the DMA Controller, perform the following:

• Select **Basic Functions > DMA > DMA Controller Intel FPGA IP** and click **Add**.

• Under the **DMA Parameters** tab, check the checkbox **Enable burst transfers** and set the **Maximum burst size** to 1024 words.

• Select the **Advanced** tab, and uncheck all the checkboxes except **doubleword**. This makes the DMA Controller allow for doubleword (64 bits) transactions only.
• Click **Finish**, then the DMA Controller `dma_0` is added to the Platform Designer system. Figure 5 shows the screenshots of the settings.

![DMA Controller Settings](image)

Figure 5. Settings for DMA Controller.

To add the Scatter-Gather DMA Controller, perform the following:

• Select **Basic Functions > DMA > Scatter-Gather DMA Controller Intel FPGA IP** and click **Add**.

• In the Configuration Wizard window, check the checkbox of **Enable burst transfers** and set the **Data width** to 32 as shown in Figure 6.

• Click **Finish**, then the Scatter-Gather DMA Controller `sgdma_0` is added to the Platform Designer system.
To add the on-chip memory, perform the following:

- Select **Basic Functions > On Chip Memory > On-chip Memory(RAM or ROM) Intel FPGA IP** and click **Add**.

- In the Configuration Wizard window, check the checkbox of **Dual-port access**. This will enable the on-chip memory to support two different clock signals.

- Set the **Slave S1 Data width** to 64 bits and the **Total Memory Size** to 8 Kbytes (8192 bytes).

- Check the check box of **Enable non-default initialization file** and set the **User created initialization file** to **onchip_memory2_0.hex** as shown in Figure 7.

- Click **Finish**, then the on-chip memory **onchip_memory2_0** is added to the Platform Designer system.
To add the Nios II processor, perform the following:

- Select **Processors and Peripherals > Embedded Processors > Nios II Processor** and click **Add**.
- Choose **Nios II/e**, which is the simplest version of the processor.
- Click **Finish** to add the Nios II Processor `nios2_qsys_0` to the Platform Designer system. There may be some error messages at the bottom of the screen, because some parameters have not been specified yet. Ignore these messages as we will provide the necessary data later.
Note that the Platform Designer automatically chooses names for the components. The name are not necessarily descriptive enough for the design, but they can be changed. To rename a component, right-click the component name and select Rename. Change the names of your components to the names shown in Figure 8.

![Figure 8. The Platform Designer system with renamed components.](image)

After adding components, you have to connected them correctly. In Platform Designer, the Connections column displays the potential connection points between components. A filled dot shows that a connection is made, while an open dot shows a potential connection point. To complete the connections in the Platform Designer system, follow these steps:
1. Connect the \texttt{pcie_ip bar1_0} Avalon\textsuperscript{®} MM master port to the \texttt{onchip_memory s1} Avalon MM slave port using the following procedures:

   (a) Right-click the IP Compiler for PCI Express component \texttt{pcie_ip}, hover in the Connections and then the \texttt{pcie_ip.bar1_0} to display possible connections.

   (b) Select the \texttt{onchip_memory.s1} on the list appeared.

2. Repeat step 1 to make the remaining connections listed in Table 2.

3. In the IRQ panel, click the connection from the Interrupt Sender of the \texttt{dma} component to the Interrupt Receiver \texttt{pcie_ip} component and type 1 into the box. Because the Platform Designer-generated IP Compiler for PCI Express implements an individual interrupt scheme, you must specify the bit to which interrupt connects. In this case, the DMA Controller’s interrupt sender signal connects to bit 1 of the IP Compiler for PCI Express input interrupt bus.

### Table 2. Complete list of Platform Designer connections

<table>
<thead>
<tr>
<th>Make Connection From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{clk_50.clk}</td>
<td>\texttt{pcie_ip.cal_blk_clk}</td>
</tr>
<tr>
<td>\texttt{clk_50.clk}</td>
<td>\texttt{onchip_memory.clk2}</td>
</tr>
<tr>
<td>\texttt{clk_50.clk}</td>
<td>\texttt{nios2.clk}</td>
</tr>
<tr>
<td>\texttt{clk_50.clk_reset}</td>
<td>\texttt{onchip_memory.reset2}</td>
</tr>
<tr>
<td>\texttt{clk_50.clk_reset}</td>
<td>\texttt{nios2.reset_n}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_clk}</td>
<td>\texttt{onchip_memory.clk1}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_clk}</td>
<td>\texttt{dma.clk}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_clk}</td>
<td>\texttt{sgdma.clk}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_reset}</td>
<td>\texttt{onchip_memory.reset1}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_reset}</td>
<td>\texttt{dma.reset}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.pcie_core_reset}</td>
<td>\texttt{sgdma.reset}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.bar1_0 (step 2)}</td>
<td>\texttt{onchip_memory.s1}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.bar2}</td>
<td>\texttt{dma.control_port_slave}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.bar2}</td>
<td>\texttt{pcie_ip.cra}</td>
</tr>
<tr>
<td>\texttt{pcie_ip.bar2}</td>
<td>\texttt{sgdma.csr}</td>
</tr>
<tr>
<td>\texttt{dma.read_master}</td>
<td>\texttt{onchip_memory.s1}</td>
</tr>
<tr>
<td>\texttt{dma.read_master}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{dma.write_master}</td>
<td>\texttt{onchip_memory.s1}</td>
</tr>
<tr>
<td>\texttt{dma.write_master}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{sgdma.descriptor_read}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{sgdma.descriptor_write}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{sgdma.m_read}</td>
<td>\texttt{onchip_memory.s1}</td>
</tr>
<tr>
<td>\texttt{sgdma.m_read}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{sgdma.m_write}</td>
<td>\texttt{onchip_memory.s1}</td>
</tr>
<tr>
<td>\texttt{sgdma.m_write}</td>
<td>\texttt{pcie_ip.txs}</td>
</tr>
<tr>
<td>\texttt{nios2.data_master}</td>
<td>\texttt{onchip_memory.s2}</td>
</tr>
<tr>
<td>\texttt{nios2.instruction_master}</td>
<td>\texttt{onchip_memory.s2}</td>
</tr>
</tbody>
</table>
4. In the **IRQ** panel, connect the Interrupt Sender of the *sgdma* component to the Interrupt Receiver *pcie_ip* component and type 2.

Besides connections inside the Platform Designer system, there are signals that have to be exported to make connections outside the system. To export signals, follow the steps:

1. In the row of the signal you want to export, click the **Export** column.
2. Accept the default name that appears in the **Export** column by clicking outside the cell without modifying the text.

Export the *pcie_ip* interfaces listed in Table 3. After the signals are connected or exported, your Platform Designer system should appear as indicated in Figure 9.

**Table 3. List of the exported interfaces**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Exported Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>refclk</td>
<td>pcie_ip_refclk</td>
</tr>
<tr>
<td>test_in</td>
<td>pcie_ip_test_in</td>
</tr>
<tr>
<td>pcie_rstn</td>
<td>pcie_ip_pcie_rstn</td>
</tr>
<tr>
<td>clocks_sim</td>
<td>pcie_ip_clocks_sim</td>
</tr>
<tr>
<td>reconfig_busy</td>
<td>pcie_ip_reconfig_busy</td>
</tr>
<tr>
<td>pipe_ext</td>
<td>pcie_ip_pipe_ext</td>
</tr>
<tr>
<td>rx_in</td>
<td>pcie_ip_rx_in</td>
</tr>
<tr>
<td>tx_out</td>
<td>pcie_ip_tx_out</td>
</tr>
<tr>
<td>reconfig_togxb</td>
<td>pcie_ip_reconfig_togxb</td>
</tr>
<tr>
<td>reconfig_gxclk</td>
<td>pcie_ip_reconfig_gxclk</td>
</tr>
<tr>
<td>reconfig_fromgxb_0</td>
<td>pcie_ip_reconfig_fromgxb_0</td>
</tr>
<tr>
<td>fixedclk</td>
<td>pcie_ip_fixedclk</td>
</tr>
</tbody>
</table>
Before you can generate the Platform Designer system, you notice that there are still some errors in the message box. Platform Designer requires that you resolve the base addresses of all Avalon MM slave interfaces in the Platform Designer system. You can either use the auto-assign feature, or specify the base addresses manually. To use the auto-assign feature, on the System menu, click **Assign Base Addresses**. Figure 10 shows the desired **Address Map**.
tab of the Platform Designer system after you have auto-assigned the base address. Make any changes needed to ensure that all addresses match those in Figure 10 before continuing.

Figure 10. The Platform Designer Address Map.

Also, you have to define the behaviour of the Nios II processor. Double-click on the nios2 component and then select onchip_memory.s2 to be the memory device for both reset vector and exception vector, as shown in Figure 11. The reset vector is the location in the memory device the processor fetches the next instruction when it is reset. Similarly, the exception vector is the memory address the processor goes to when an interrupt is raised.

Figure 11. The Settings for Nios II processor.

Now, the error messages should disappear and your Platform Designer system is finished. To generate the system, perform the following:
1. Select Generate > Generate HDL, uncheck the Create block symbol file (.bsf) in the Synthesis section as shown in Figure 12.

2. Click the Generate button at the bottom of the tab.

3. After Platform Designer reports Generate Completed in the Generate progress box title, click Close.

4. On the File menu, click Save and then close the Platform Designer application.

![Figure 12. The Platform Designer Generation tab.](image)

### 3.2 Adding PLL Using IP Catalog

Besides the Platform Designer system, you need to add a Phase-locked loop (PLL) block, which is not added as a Platform Designer component. The PLL block is used to provide different clock signals for the Platform Designer system. To add the PLL block, perform the following:

1. Select Tools > IP Catalog.

2. In the pop-up panel, select Library > Basic Functions > Clocks; PLLs and Resets > PLL > ALTPLL and click Add... button.

3. Choose the Verilog as the output file type for your design, and specify a variation name for output files. For this walkthrough, specify my_pll.v as the name of the IP core file: <working_dir>/my_pll.v, as shown in Figure 13, and click OK.
4. Under the **Parameter Settings** tab, set **What is the frequency of the inclk0 input** to 50MHz as shown in Figure 14 and click **Next** to go to the **Inputs/Lock** section.

5. Uncheck the checkboxes of **Create an ‘areset’ input to asynchronously reset the PLL** and **Create ‘locked’ output** as shown in Figure 15.
6. Under the **Output Clocks** tab, enter 50MHz after selecting **Enter output clock frequency** as shown in Figure 16, and click **Next** to go to the **clk_c1** section.
7. Select the checkbox of **Use this clock** and then set the output clock frequency to 125MHz with the same procedures of the last step, and click **Next** to go to the **clk_c2** section.

8. Select the checkbox of **Use this clock** and then set the output clock frequency to 100MHz with the same procedures of the last step.

9. Under the **Summary** tab, uncheck **my_pll_bb.v** as shown in Figure 17, and then click **Finish** to generate the file.
3.3 Integrating modules into the Quartus® Prime Project

To complete your hardware design, perform the following:

1. Type the code in Figure 18 into a file called `de4_pcie_tutorial.v`. This code is a top-level Verilog module that instantiates the Platform Designer system and the PLL block. The module is named `de4_pcie_tutorial`, because this is the name we specified in Figure 2 for the top-level design entity in the Quartus Prime project.
module de4_pcie_tutorial(
    input OSC_50_BANK2,
    input PCIE_PREST_n,
    input PCIE_REFCLK_p,
    input [3:0] PCIE_RX_p,
    output [3:0] PCIE_TX_p,
    output [7:0] LED
);

assign LED[7] = 1'bl;
assign LED[3] = 1'bl;
wire clk50, clk125, clk100;
my_pll pll_inst(
    .inclk0 (OSC_50_BANK2),
    .c0 (clk50),
    .c1 (clk125),
    .c2 (clk100)
);

qsys_system system_inst(
    .clk_clk (clk50),
    .pcie_ip_refclk_export (PCIE_REFCLK_p),
    .pcie_ip_fixedclk_clk (clk125),
    .reset_reset_n (1’b1),
    .pcie_ip_PCIE_RSTN_export (PCIE_PREST_n),
    .pcie_ip_PCIE_RX_DATAIN_0 (PCIE_RX_p[0]),
    .pcie_ip_PCIE_RX_DATAIN_1 (PCIE_RX_p[1]),
    .pcie_ip_PCIE_RX_DATAIN_2 (PCIE_RX_p[2]),
    .pcie_ip_PCIE_RX_DATAIN_3 (PCIE_RX_p[3]),
    .pcie_ip_PCIE_TX_DATAOUT_0 (PCIE_TX_p[0]),
    .pcie_ip_PCIE_TX_DATAOUT_1 (PCIE_TX_p[1]),
    .pcie_ip_PCIE_TX_DATAOUT_2 (PCIE_TX_p[2]),
    .pcie_ip_PCIE_TX_DATAOUT_3 (PCIE_TX_p[3]),
);
endmodule

Figure 18. Verilog code for the top-level module

2. Add the `de4_pcie_tutorial.v` file, the `qsys_system.qip` file, the `my_pll.qip` file, and the `onchip_memory2_0.hex` file to your Quartus Prime project as shown in Figure 19. The `onchip_memory2_0.hex` file is used to initialize the on-chip memory and can be found in the directory `design_files`, which can be downloaded along with this tutorial from the Intel FPGA University Program website. It contains the executable file of the demo application for the Nios II processor, which will be used in the next section.

3. Add the necessary pin assignments on the DE4 board to your project. Note that an easy way of making the pin assignments when you use the same pin names as in the DE4 User Manual is to import the assignments from file. The pin assignments can be found in the `DE4_230_pin_assignments.qsf` or the `DE4_530_pin_assignments.qsf` file (depending on your board), in the directory `design_files`. 

Intel Corporation - FPGA University Program

March 2019
4. Compile the Quartus Prime project.

3.4 Programming and Configuration

Before you can configure the DE4 board, you should plug the DE4 board into the PCI Express port of your Linux computer and connect the board to the computer by means of a USB cable plugged into the USB-Blaster port. Then turn on the power of the board and program the FPGA in the JTAG* programming mode as follows:

1. Select **Tools > Programmer** to reach the window in Figure 20.

2. If the USB-Blaster is not chosen by default, press the **Hardware Setup...** button and select the USB Blaster in the window that pops up.
3. The configuration file `de4_pcie_tutorial.sof` should be listed in the window. If the file is not already listed, then click Add File and select it from the `output_files` subdirectory of the project.

4. At this point the window settings should appear as indicated in Figure 20. Press Start to configure the FPGA.

After you have successfully configured the hardware in the FPGA device, you should reboot your Linux computer to let the operating system to detect the PCI Express device you built. Ensure that the DE4 board is powered.

After the reboot, type command `lspci` in the terminal to ask the operating system to list all the PCI devices on the computer. You should be able to see an Intel `0xde4` device shown on the terminal.

### 4 Using the Driver

In the previous section, you have built a hardware system that supports the PCI Express; however, you need a driver to use the PCI Express to communicate with the DE4 board. Intel FPGA University Program provides an open source PCI driver for the usage of DE4 board in Linux operating system. The driver is designed to meet the general needs of using the PCI Express. It allows a user to:
1. create a configuration file to define the hardware specific information for the driver.

2. access any Base Address Registers (BAR).

3. use DMA Controller and Scatter-Gather DMA Controller.

4. use either polling or interrupt for DMA transfers.

The driver is in the directory design_files/driver. The driver is tested on Linux 2.6.32, but it should work with newer versions of the Linux kernel.

4.1 Creating the Configuration File

Before you can use the driver, you need to create a file to configure the driver for your hardware system. An example configuration file, config_file_example, can be found in the driver folder. To make the file for the system you built, open config_file_example and perform the following:

1. Set the Vendor ID and Device ID. They should be the same with the IDs shown in Figure 4.
   
   vendor_id = 0x1172
   
   device_id = 0x0de4

2. Set the parameters related to the PCI Express IP Core.

   pci_dma_bit_range = 31 This defines the bit mask for DMA buffers. By setting it to 31, you force Linux to allocate physical address between 0x00000000 – 0x7FFFFFFF for DMA buffers. This parameter is decided by the settings of Address Translation in the IP Compiler for PCI Express.

   tx_base_addr = 0x80000000 The base address of the txs Avalon MM slave.

   pcie_cra_bar_no = 2 This defines the number of the BAR which is connected to the cra Avalon MM slave.

   pcie_cra_base_addr = 0x00000000 The base address of cra Avalon MM slave.

3. Set the parameters for DMA controllers. By default, the driver supports at most four DMA controllers, so there are four columns in the config_file_example. You only need to fill in the first two columns for the DMA Controller and the Scatter-Gather DMA Controller in your system, and leave other columns as zeros.

   dma_type = 1, 2 This defines the type of the DMA controller. The DMA Controller is represented by 1, while the Scatter-Gather DMA Controller is represented by 2.

   dma_irq_no = 1, 2 This defines the interrupt number of DMA controller as shown Figure 9.

   dma_ctrl_bar_no = 2, 2 This defines the number of the BAR, which is connected to the control port of the DMA controller.

   dma_ctrl_base_addr = 0x00004000, 0x00004040 This defines the base address of the control port.

4. Set the data width of the allowed transactions for DMA Controller. This is necessary if you are using the DMA Controller.
sdma_data_width = 8, 0  Set the first number to 8 (words) to match the settings in Figure 5 and ignore it for the Scatter-Gather DMA Controller.

5. Save the file as tutorial_config_file.

4.2 Load the Driver Module into the Kernel

To install the driver, you need to load the .ko module file into the Linux kernel. If it’s your first time to use the driver, you have to compile the driver codes into the .ko module file. Run the makefile in the folder driver to compile the driver. Note that you may have to change the file format from the Windows* Format to the UNIX* Format first, so go to the demo and driver folders before compiling any programs, type:

```
dos2unix *
```

After you compiled the driver, you will get the kernel module file alt_up_pci.ko. The name of the module file is determined by the makefile and you are recommended not to change the name. Once you have the .ko module file, you don’t have to recompile the driver for your future usage of the driver.

Each time you use the driver, you will have to dynamically load the driver module with the correct configuration file. This is done by a shell script load_alt_up_pci_driver.sh provided in the driver folder. Load the driver module with the configuration file you created by typing:

```
sudo ./load_alt_up_pci_driver.sh tutorial_config_file
```

Note that you need to pass the file name of the configuration file as the first argument to the shell script, otherwise the script will use config_file as default. If you have changed the makefile when compiling the driver codes, you have to modify the variable DRV_MODULE in the shell script to match the name of your .ko module file.

This script requires that your account has the administrator right. After typing in the password, you will see Matching Device Found on success, or Matching Device Not Found when failed. If the module is successfully loaded, the script will automatically create a character special file alt_up_pci0 in directory /dev/. This file is important because it represents the DE4 device and you will need to access it when you want to use the driver.

Since the driver is loaded into the kernel, the information printed out by the driver will not show on the terminal window directly. Instead, they will be printed as driver messages. To see the driver messages, type command:

```
dmesg
```

The driver messages are important messages to check the status of the driver especially when you encounter problems. You should be able to see similar driver messages shown in Figure 21 after you have loaded the driver module successfully.
If you want to remove the module from the kernel, you can use the shell script below:

```
sudo ./unload_alt_up_pci_driver.sh
```

### 4.3 How to Use the Driver

To help you understand how to use the driver, a demo application is provided. The application will ask you to input a sentence and the sentence will be sent to the on-chip memory of the DE4 board. Then the Nios II processor will change the sentence, modifying letters from lower-case to upper-case and vice versa. After the processing, the application will read the sentence back and print it on the screen. To ensure the data is shared correctly between the PC host and the DE4 board, one byte (0x00000FFF) in the on-chip memory is used as a flag to indicate who owns the control of the data. ‘H’ means the data is owned by the host while ‘B’ means it’s owned by the board. When the data is owned by one side, the other side should do nothing other than polling the control byte.

Figure 22 shows the simplified code of the demo application, which does not contain error checking. You can find the complete code `demo.c` in directory `design_files/demo`. 

![Figure 21. Messages shown on the terminal](image)
```c
#include <stdio.h>
#include <string.h>
#include "alt_up_pci_lib.h"

#define ONCHIP_CONTROL 0x00000FFF
#define ONCHIP_DATA 0x00001000
#define MAX_DATA_SIZE 4096
#define CTRLLER_ID 0

int main() {
    int fd, length_str;
    char buff[MAX_DATA_SIZE], control;

    alt_up_pci_open( &fd, "/dev/alt_up_pci0" );

    control = 'H';  // controlled by the host PC
    alt_up_pci_write( fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

    while(1) {  
        fgets(buff, MAX_DATA_SIZE, stdin);
        length_buff = strlen(buff) + ( 8 0 - (strlen(buff)%8));

        alt_up_pci_dma_add(fd, CTRLLER_ID, ONCHIP_DATA, buff, length_str, TO_DEVICE);
        alt_up_pci_dma_go (fd, CTRLLER_ID, INTERRUPT);

        control = 'B';  // pass the control right to the DE4 board
        alt_up_pci_write (fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

        while( control != 'H')  // polling the control byte
            alt_up_pci_read( fd, BAR0, ONCHIP_CONTROL, &control, sizeof(control) );

        alt_up_pci_dma_add(fd, CTRLLER_ID, ONCHIP_DATA, buff, length_str, FROM_DEVICE);
        alt_up_pci_dma_go(fd, CTRLLER_ID, INTERRUPT);

        printf("Received : \n%s\n", buff);
    }

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

Figure 22. Simplified code for demo application

As you can see in Figure 22, there are six functions prefixed with `alt_up_pci_`. These functions are provided in the header file `alt_up_pci_lib.h`. The descriptions of the functions are shown below. Note that all these functions will return 0 on success, and return –1 when failed.
**alt_up_pci_open()** – This function is used to open the DE4 device file.

You will have to input the character special file of the device, which in this case is `alt_up_pci0`, then you will get a file descriptor. After calling this function successfully, you can start using other functions to perform operations on the DE4 board.

**alt_up_pci_close()** – This function used to close the DE4 device file.

You will have to pass the file descriptor got from `alt_up_pci_open()` to the function. This function should be called at the end of the application.

**alt_up_pci_read()** – This function is used to do the read operation.

You will have to select which BAR to read from and what is the starting address of the read operation. Also, you need to pass a pointer to the buffer and the size of the buffer to the function.

**alt_up_pci_write()** – This function is used to do the write operation.

This function is similar to `alt_up_pci_read()` and does the write operation instead.

**alt_up_pci_dma_add()** – This function is used to add a DMA transfer into the queue of the DMA controller.

Inside the driver, there is a queue for each DMA controller. You are allowed to push up to ten DMA transfers into the queue before the DMA controller really performs the transfers. You will need to pick a DMA controller to perform the transfer, and give it the pointer to the buffer, the address in the DE4 board, how many bytes you want to transfer and the direction of the transfer. Note that the driver will not check the overflow of the address, so the application is responsible to ensure that the DMA controller is not reading from or writing to an illegal address.

**alt_up_pci_dma_go()** – This function is used to start the DMA transfers in the queue.

By calling this function, the selected DMA controller will start performing all the DMA transfers in the queue, and it will use either polling or interrupt to check whether a transfer is finished.

The header file `alt_up_pci_lib.h` also contains three enum types for the BARs, the direction of the DMA, and the method of checking to increase the readability of the codes. We recommend you to include this file when you are writing your own application.

Pay attention to the transfer length when using the DMA Controller, because the data length have to match the settings you made for the DMA Controller. In this tutorial, an error will occur, if the length is not a multiple of 8 or the address is not aligned with 8 bytes.

### 4.4 Running the Demo Application

To try the demo application, compile and run the demo codes in the demo folder. By typing sentence into the terminal and typing Enter to send the sentence, you can see how the sentence is changed by the Nios II processor. You can also change the value of `CTRLLER_ID` in the `demo.c` to select a different DMA controller.

The code shown in Figure 23 is the code run by the Nios II processor. It is downloaded to the on-chip memory by adding the `onchip_memory2_0.hex` during the compile time.
```c
#include <stdio.h>

volatile char *onchip_control = (char *) 0x00000FFF;
volatile char *onchip_data = (char *) 0x00001000;

int main() {
    int i;
    char ch;

    while ( 1 ) {
        while ( *onchip_control != 'B' ) // polling the control byte
            ;

        i = 0;
        while ( (ch = *(onchip_data + i) ) != '\n' ) {
            if ( ch >= 'a' && ch <= 'z' )
                *(onchip_data + i) = ch - 'a' + 'A';
            else if ( ch >= 'A' && ch <= 'Z' )
                *(onchip_data + i) = ch - 'A' + 'a';
            else
                *(onchip_data + i) = ch;
            i++;
        }
        *onchip_control = 'H'; // pass the control right to the host PC
    }
    return 0;
}
```

Figure 23. Program run by the Nios II Processor

4.5 Reprogramming the FPGA without Rebooting the Host Computer

After you successfully configure the FPGA and install the driver on your host computer, you may still want to modify your PCI Express settings on the FPGA and reprogram it. However, reprogramming the FPGA will disable the connection between the FPGA and the host computer. It requires you to reboot the host computer and reload the driver every time you download a new configuration onto the FPGA, which is inconvenient and time-consuming. Here we provide a way to reprogram the FPGA without rebooting the host computer if a valid PCI Express connection has already been established:

1. To save the PCI Express registers, go to the demo folder, run:

   ```shell
   ./reprogram save
   ```

2. Program a new image(.sof) on FPGA following the instructions in section 3.4. Note that the memory space, the Vendor ID and the Device ID of the PCI Express shouldn’t be changed in Platform Designer, otherwise restoring the register information will give the wrong information to the host computer.
3. Then we need to restore the registers information, go to the demo folder, run:

   ```
   ./reprogram restore
   ```

4. Go to the driver folder to unload and reload the driver using:

   ```
   ./unload_alt_up_pci_driver.sh
   ./load_alt_up_pci_driver.sh tutorial_config_file
   ```

The device connection should be re-established.

### 5 Changing the Driver

Now, you can start building your own system and applications. However, if you are not satisfied with the functionality provided by the driver and want to design your custom driver, you can read the `index.html` in directory `design_files/doc/html`. This file is generated by the Doxygen and it will give you a general idea of the driver, so that you can change it.
Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel’s standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

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