Extra info / appendix

1. Relationship of hardware and software in the DE2-115 system

The digital design is provided in the form of a **.sof** programming file. To program the Cyclone IV on the DE2-115, start **Quartus** from the **Altera** submenu of the Start menu. Connect the DE2-115 board with power and ensure the USB cable is connected to the port labelled "BLASTER" on the board. Select "**Programmer**" from the "**Tools**" menu to open up the programmer as shown in Figure 1.1. Click the "**Hardware Setup**" button to make sure the currently selected hardware is **USB Blaster** as shown in Figure 1.2. If this is not the case, select **USB-Blaster** from the **Available hardware items** list, click the "**Add Hardware**" button and close the dialog. If **USB-Blaster** is not present in the list, then the board may not be connected properly or your computer may not have the correct drivers installed. Click on the "Add File" or "Change File" button on the left of the Programmer dialog to include "freq_relay_controller.sof" and tick the box under "**Program/Configure**". Then click the "**Start**" button to download the hardware design to the DE2-115 board. The progress of the download is displayed on the progress bar. Now you are ready to start the software implementation of the Nios II system.
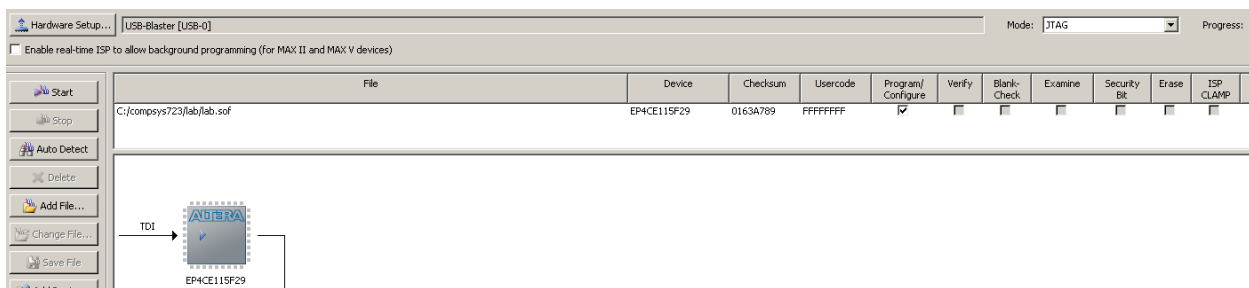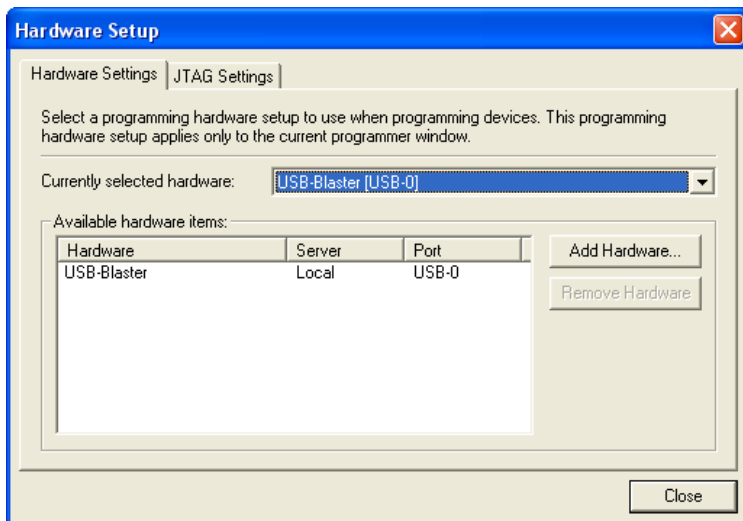

Figure 1.1: Programmer of Quartus II

Figure 1.2: Hardware settings of the download cable

Introducing software to the system

The **hardware design** is ready and downloaded to the FGPA, however, the Nios II processor must be loaded with a program to execute.

2. Software implementation on the DE2-115 with dedicated hardware design

Developing software with Nios II IDE

Software development for the Nios II platform can be performed using the Nios II Software Build Tools IDE (integrated development environment) which is based on Eclipse. Start the Nios II IDE from the start menu. The main IDE window is illustrated in Figure 2.1.
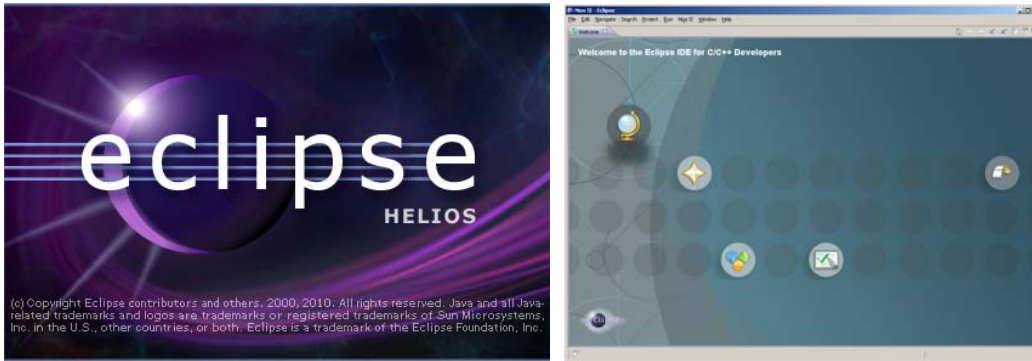

Figure 2.1: Nios II IDE

Construct an application with basic BSP

To use the Nios II IDE, you must first create a **workspace**. A workspace is used to store the current state of the IDE. It stores the currently opened projects and their associated configurations. When opening Eclipse for the first time, it will ask you to specify a workspace directory. You can also change the current workspace to an existing workspace or a new workspace using the File menu (**File->Switch Workspace**).

We will create a new workspace for this lab. Browse and create a directory (named **software**) in the directory where you have placed the provided **freq_relay_controller.sof** and **nios2.sopcinfo** files and use this as the workspace directory. For example, the path "c:\compsys723\lab\software" is used in Figure 2-2.
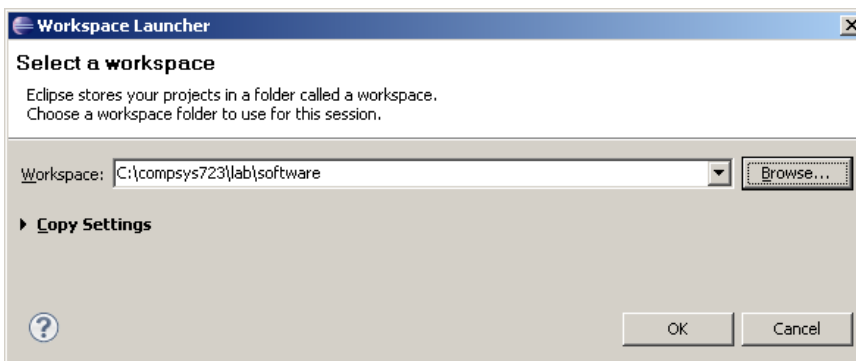

Figure 2.2: Switch workspace to the software directory

To create an application, select "**File->New->Nios II Application and BSP from Template**" from the main menu and perform the following steps as shown in Figure 2.3:
1. Choose nios2.sopcinfo for the SOPC Information File name.
2. Choose **cpu** for the **CPU name**.
3. Input hello_lab for the Project name.
4. You should see "c:\compsys723\lab\software\hello_lab" appear as the default project location.
5. Choose "Hello World" for the Project template.
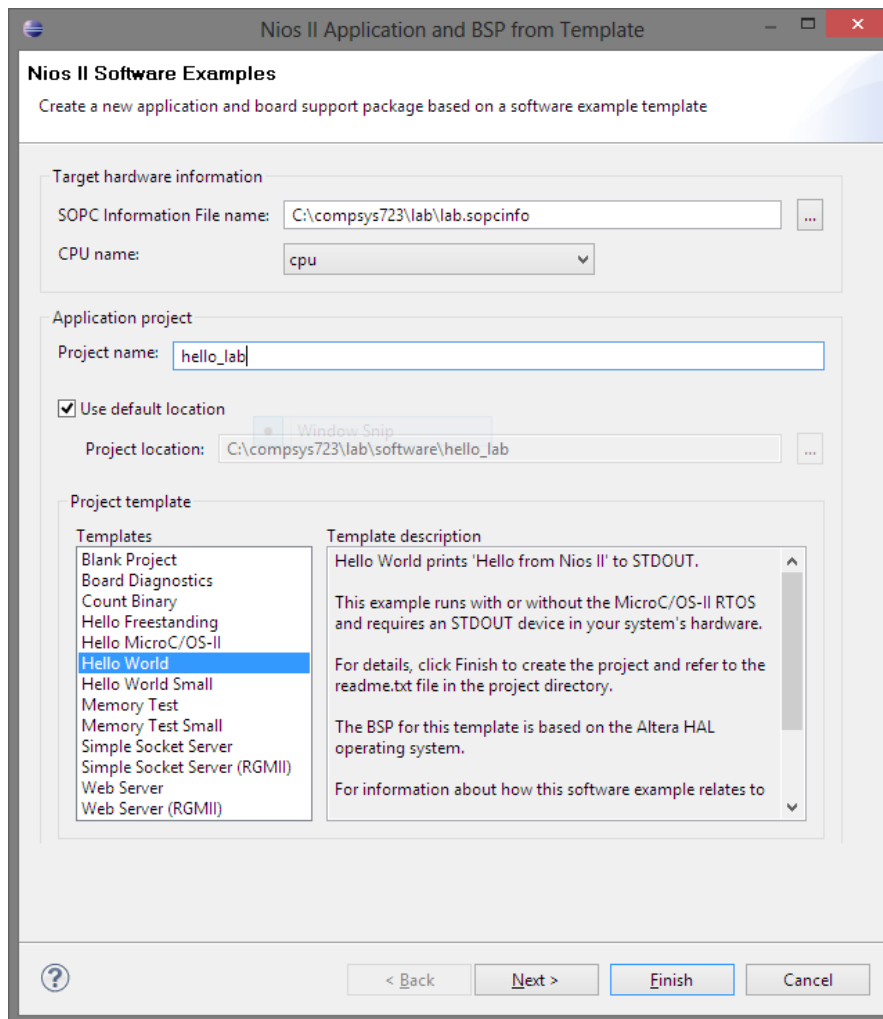6. Click "Next".

Figure 2.3: Setting of the new project

As shown in Figure 2.4, use "hello_lab_bsp" as the name of the BSP and use the default location provided. Click "Finish" to generate the template Nios II application and corresponding BSP.
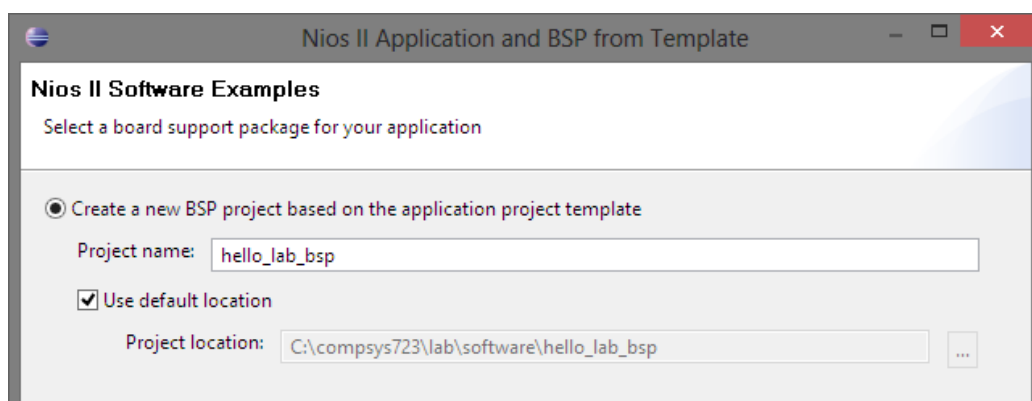

Figure 2.4: Creation of BSP

We have to specify where the program code will be stored (this can either be on-chip memory, SDRAM, Flash memory, or SRAM). The relevant settings are shown in Figure 2.5:

1. Right click on the BSP project named **hello_lab_bsp**.
2. Click on **Properties**.
3. Click on **Nios II BSP Properties** in the left pane.
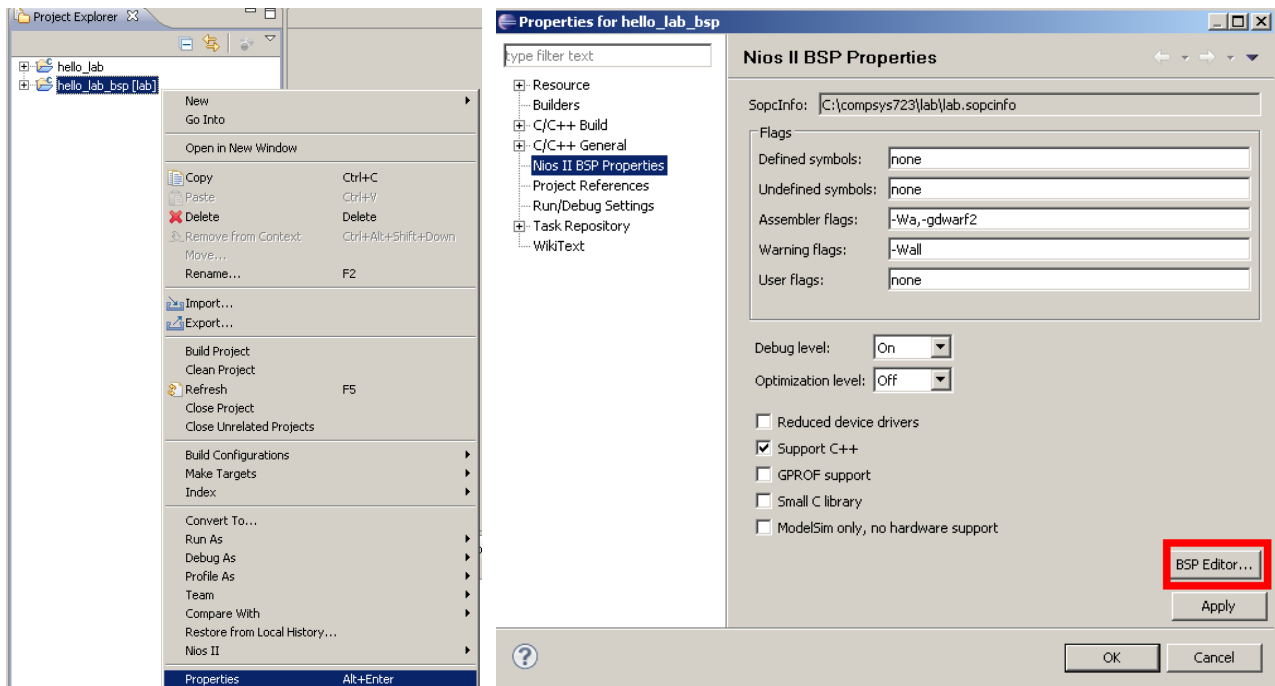4. Select BSP Editor.

3

Figure 2.5: Open the BSP Editor menu

The settings of the BSP will be displayed, as shown in Figure 2.6. The settings tabs include the following:

1. Main: options which configure the HAL, including the devices to use for standard input, output, and error output of the system.
2. Software packages: optional software packages required for the application, this includes packages which enable the use of compressed file systems.
3. Drivers: drivers provided by Altera and other vendors are listed here.
4. Linker Script: configuration of the linking phase of the compilation process in which the object files of the project are combined into a single executable which is loaded into memory.
5. Enable File Generation: configuration of the by-product files which will be generated along with the BSP
6. Target BSP directory: the structure of the BSP directory that will be generated.

We will configure a number of settings in the **Main** and **Linker Script** tabs and keep the default settings in the other tabs.
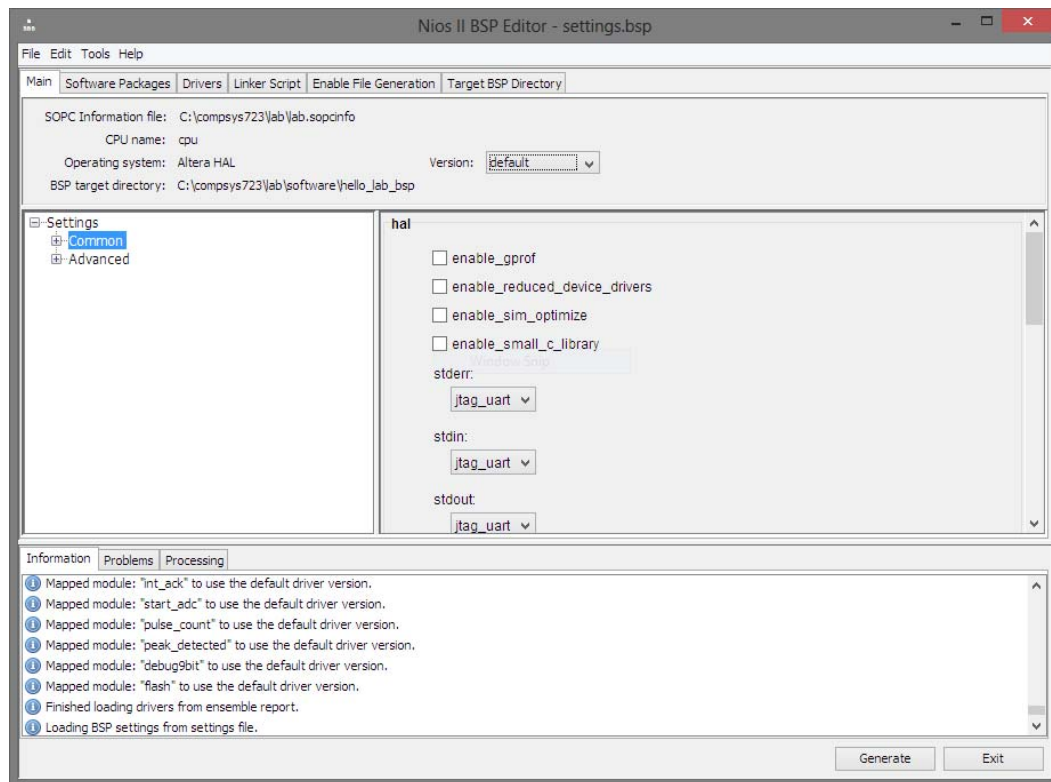
Figure 2.6: Settings of the BSP

In the **Main** tab, choose "**Settings**->**Common**" in the left pane and follow the steps below which are also illustrated in Figure 2.7:
1. Select "**jtag_uart**" for **stderr**, **stdin**, and **stdout**. You might want to change these options in future depending on your application, but we will leave it as the default (jtag_uart) for now. This will cause output from the program to be shown in the console in the Nios II IDE or the terminal program that is provided (named: **nios2-terminal**). There are a number of character output devices which can be selected, including *uart*, *character_lcd* (the LCD on the board), or *jtag_uart* (a shared connection with the USB-BLASTER which uses the JTAG for communication).
2. Select "**timer_1ms**" for the **sys_clk_timer**. This changes the frequency of the system timer to 1 KHz.
3. Select "**sdram**" for:
    1. exception_stack_memory_region_name
    2. interrupt_stack_memory_region_name

This uses the SDRAM for the stack when interrupts or exceptions occur while running a program.

In the **Linker Script** tab, make sure "**sdram**" is selected as shown in Figure 2.8 for the following fields:
1. **.bss**: stores uninitialized statically allocated variables. This includes all global arrays and variables as well as static variables declared locally in a function, which are not initialized with a value.
2. **.heap**: the storage area for memory that is dynamically allocated using the **malloc** function call. Dynamically allocated memory can be useful for implementing many data structures such as linked lists.
3. **.rodata**: is used for constant, read-only data such as constant values that are defined with the *define* macro. For instance, *#define ABC 100* will store a value of 100 in this section.
4. **.rwdata:** stores initialized global and static variables which may be modified during program execution. For example, a local static variable with the following declaration: *static int test = 20*, will cause the "test" variable to be stored in this section, with an initial value of 20.
5. **.stack**: this section is used for storing non-static local variables of a function. This includes the arguments passed to the function.

Click "Generate" to populate the settings and click "Exit" to return to the previous properties window shown in Figure 2.5.
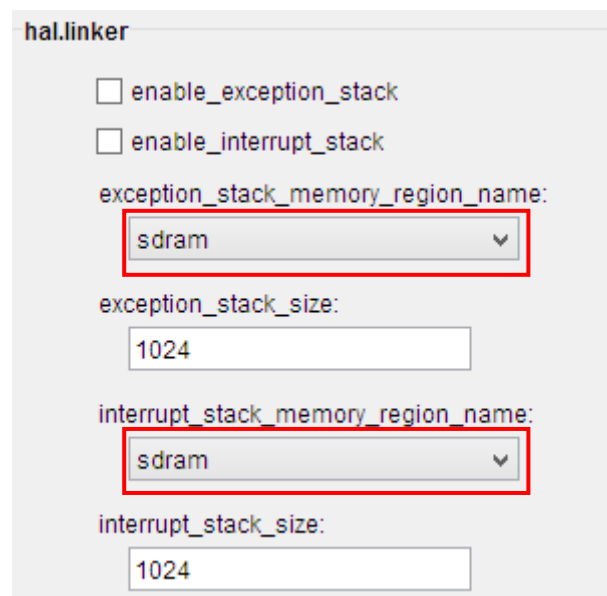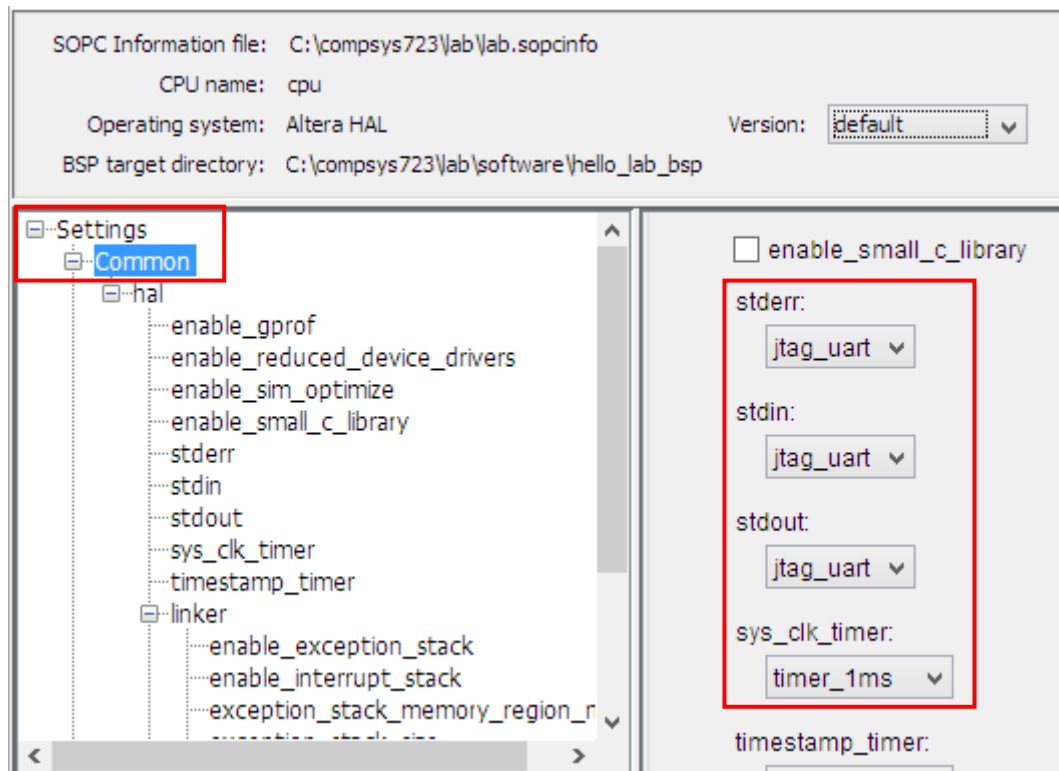
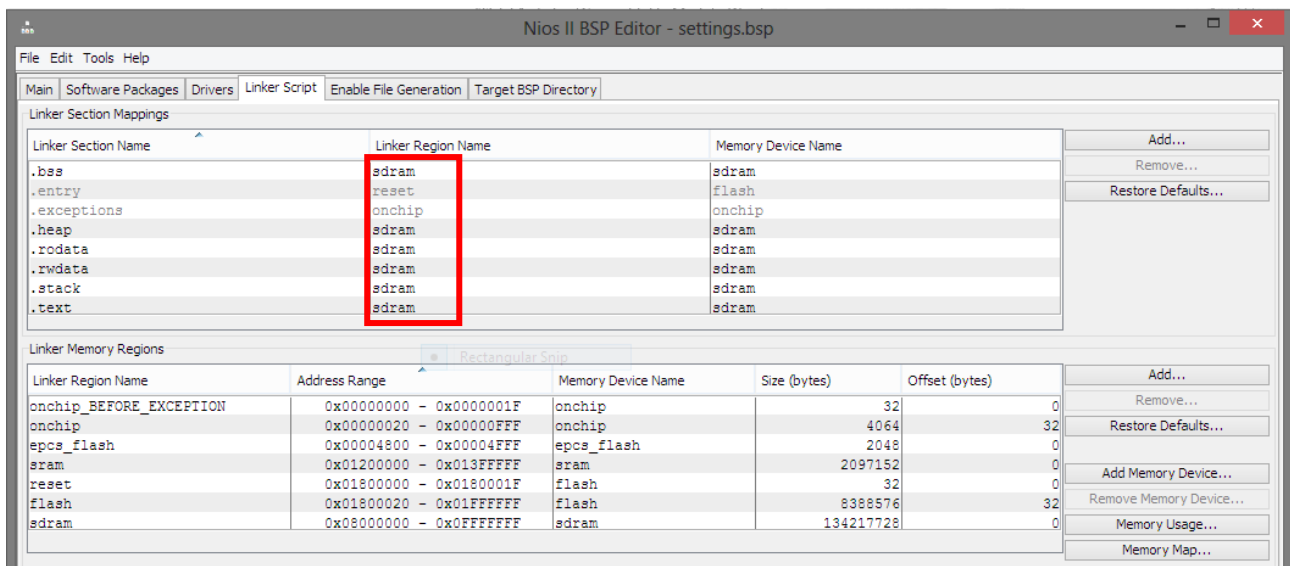Figure 2.7: Settings in the Main tab

6

Figure 2.8: Settings in the Linker Script tab

The other options in the BSP editor allow you to further customize the BSP for specific needs. For example, a developer would want to tick "**Small C library**" and/or un-tick "**Support C++**" to limit the memory usage on a system with a very constrained amount of available memory.

3. Compile and run the first Nios II program in this lab

To build and run the Nios II program, perform the following:

1. Right-click on the **hello_lab** project and select "**Build Project**" to start compiling, as shown in Figure 3.1. You don't need to do this for the BSP project as it is a dependency of the hello_lab project, so the BSP project will be built automatically if necessary (for instance when the BSP settings are modified).

2. To run the application, click "**Run**" from the menu bar and select "**Run Configurations**". **Right-click** on "**Nios II Hardware**" and select "**New**" as shown in Figure 3.2.

3. Make sure the **Project** tab is selected and use the name "hello_lab" for this configuration. Make sure the **project name** is **hello_lab** by selecting it from the dropdown box.

4. Now change to the "**Target Connection**" tab. As shown in Figure 3.4, select "**USB-Blaster**" for the **Processor** and **Byte Stream Device**. If **USB-Blaster** is not displayed, make sure the board is connected to the computer with the provided SOF file correctly downloaded onto it and click **Refresh**. Now click **"Run"** on the lower right corner. If code in the project has changed, the project may be re-built before being loaded onto the processor.

5. You should soon be able to see the progress of the building and downloading process in the Console tab of the IDE as shown in Figure 3.5. *"Hello from Nios II"* will be displayed in the "Nios II Console" as shown in Figure 3.6, once the application has been downloaded and executed successfully. Click 🔳 to terminate the execution.
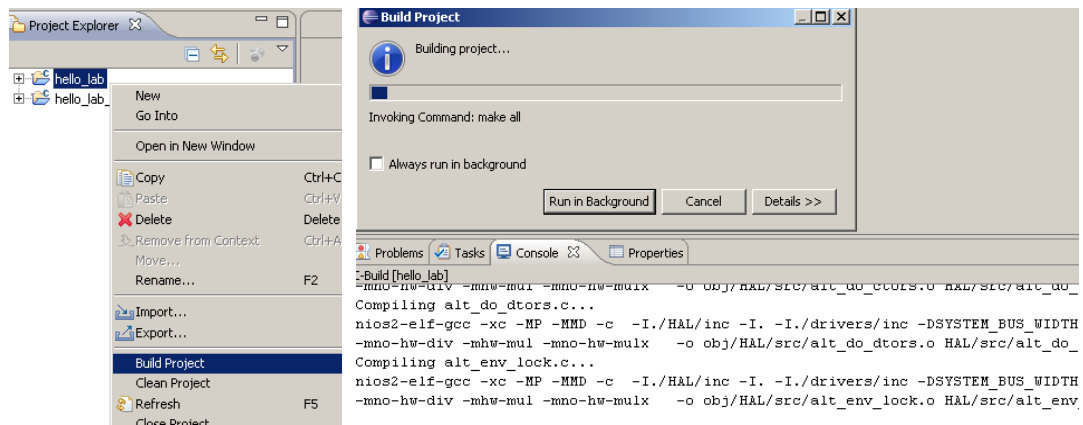
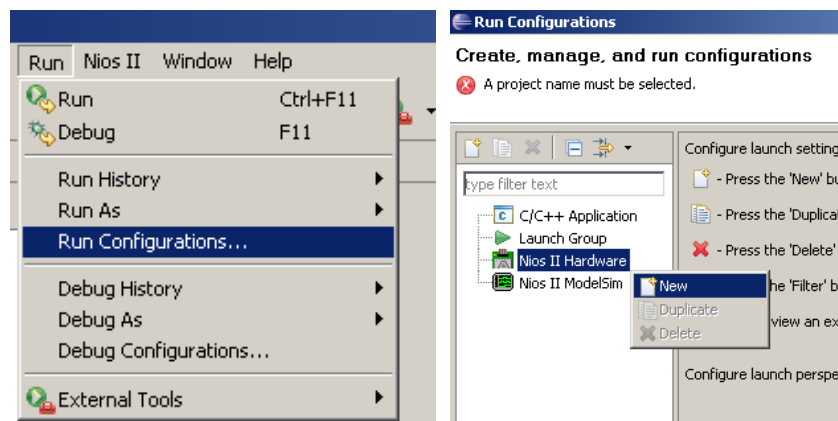Figure 3.1: Building the hello_lab application

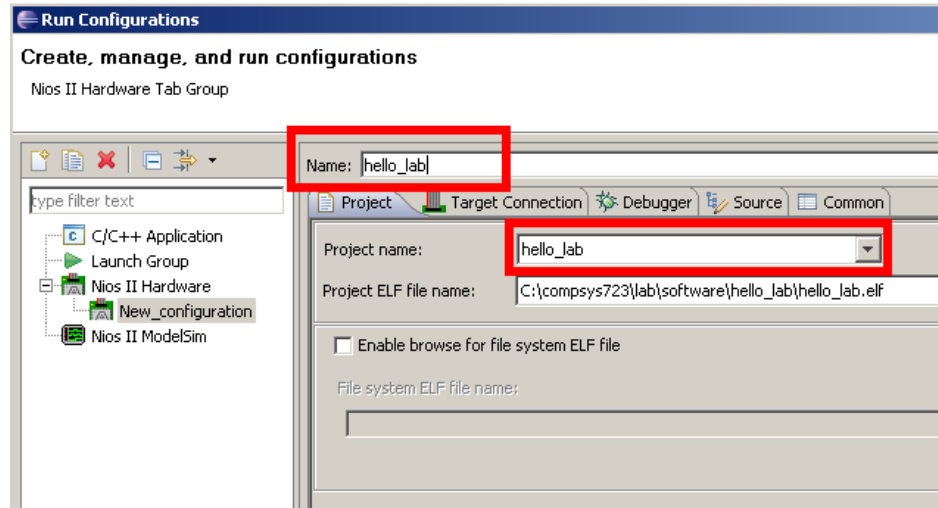

Figure 3.2: Running on the Nios II hardware

Figure 3.3: The Project tab of configuring application execution on Nios II
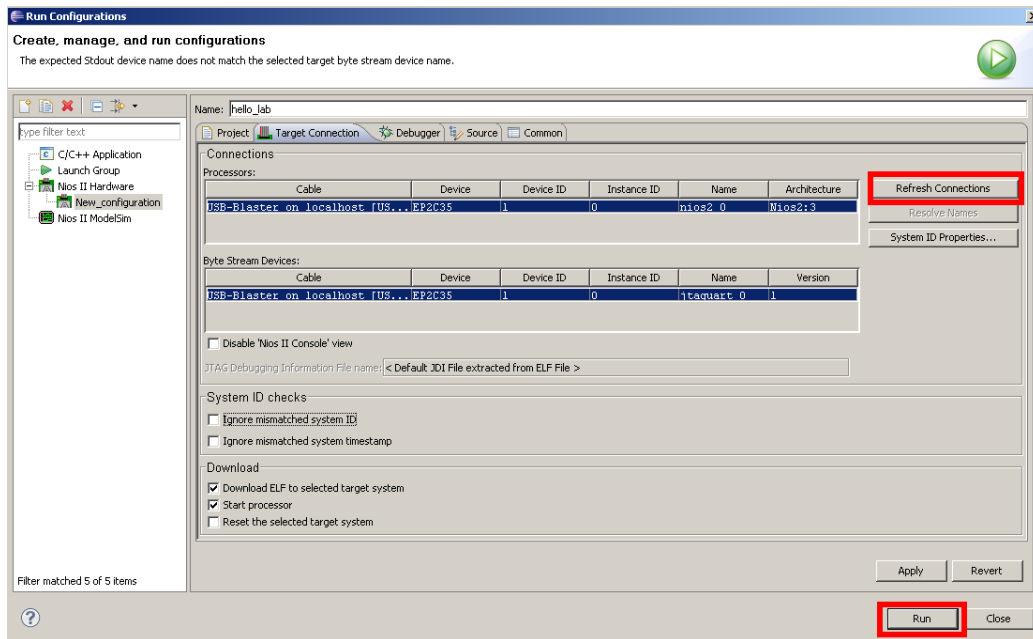
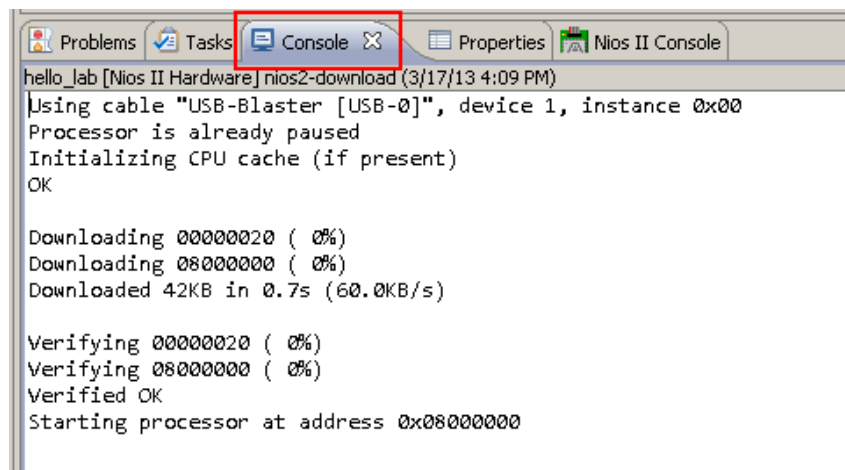Figure 3.4: The Target Connection tab of configuring application execution

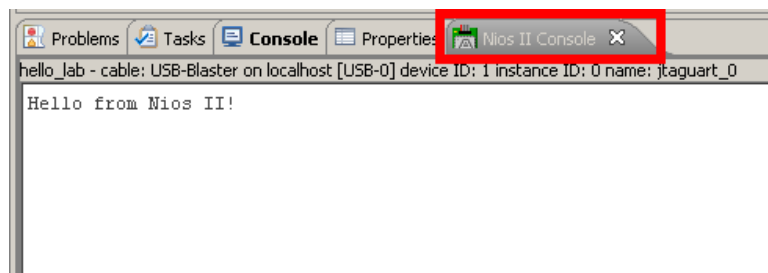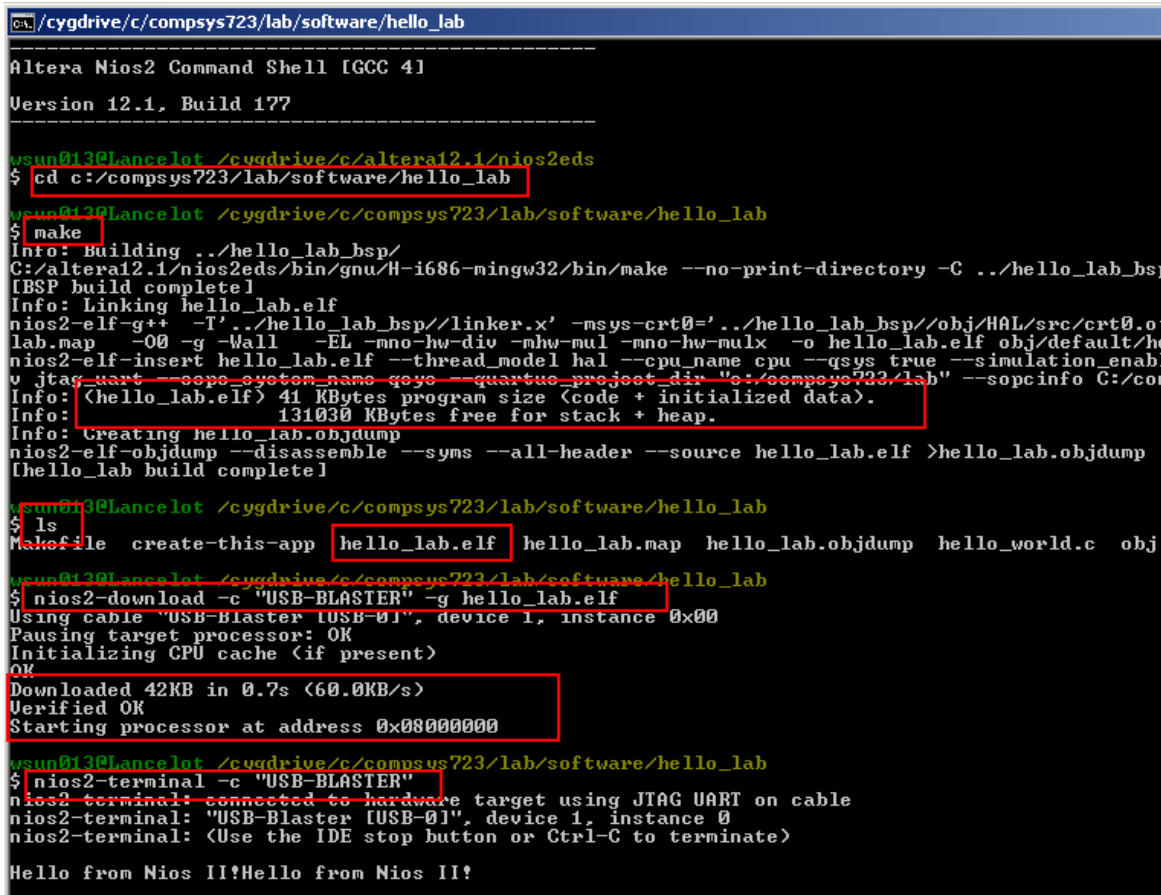Figure 3.5: Displaying messages of downloading application to Nios II in the Console tab

Figure 3.6: The output from the hello_lab application in the Nios II console

4. Nios terminal and some other commands

Sometimes, working in a command-line environment can be very handy. To do so, perform the following, as shown in Figure 4.1:

1. Launch the "**Nios II Command Shell"** from the Altera folder in the start menu, or browse to the installation directory of the Nios II development tools (for instance, C:\altera12.1\nios2eds) and double click "**Nios II Command Shell.bat**".

2. Change the current directory to the location where your application resides (**cd c:\compsys723\lab\software\hello_lab**), and then type "**make**" to build the application. If the build succeeds, you should be able to see a file (by using **ls** or **dir**) named *your_project_name.elf* (**hello_lab.elf** in this case).
3. Download the program and start running it by using the following command: **nios2-download –c "USB-BLASTER" -g hello_lab.elf**. Launch a terminal program to monitor the standard output by executing **nios2-terminal**. Note that you need to make sure the console is closed in the Nios II IDE before running this command. If the project has been launched in the Nios II IDE, click on the 🔴 button to terminate the execution. This releases the jtag_uart, allowing it to be used by nios2-terminal.

Figure 4.1: Using Nios II tools under command line environemnt

From Figure 4.1, you can observe the following:
1. The BSP is built first, before the application
2. The size of the Nios II application (a program size of 41 KB and 131030 KB (128MB) of free memory in the SDRAM for the stack and heap).
3. The address where the processor begins execution (0x08000000). This is usually the address of a reset exception vector which will then jump to the **entry-point** of the program code which is to be executed.

More insight into peripherals

Peripherals provided by Qsys are components that are designed to follow a specification which ensures compatibility with the **Avalon Bus**. The Avalon Bus is the ***glue-logic which interconnects the components in a system using numerous arbitrators and multiplexers***. Avalon Bus compatible components can be further divided into two groups – **master** and **slave** components. The Nios II processor is an example of a master component. LEDs, switches, push buttons, timers, and UART peripherals are examples of slave components. One of the major differences between masters and slaves is, a master can issue commands to other components as well as receive commands from other masters. Slaves can only receive and process commands issued by a master.

Slaves are often designed to contain **internal registers**. These registers are used to help the slave complete certain tasks. Configurations, coefficients, or debugging information can be stored in these registers. During operation, slaves read the configuration and act accordingly. In this lab, the registers in the slaves are used for configuration.

Even though they are called registers, they are different to the registers (sometimes called the register file) within the processor. Processor registers can be access directly. However, registers of slave components are **memory mapped** in the processor's address space (the Nios II in this case). The organization of the Nios II processor and peripherals is shown in Figure 4.2.

Slave registers are accessed through memory access operations. The slave registers are mapped in the processor's address space according to the addresses specified in Qsys (which is out of the scope of this lab). For instance, as shown in Figure 4.2, two slave components are added with memory addresses 0x00001000 and 0x00002000 respectively. Generally speaking, registers within the slave components can be of variable length but in practice they are usually designed to be the same width as the processor data bus/internal register which is 32-bits for the Nios II. The address of the system is **byte**-based, where the minimum size of data that can be accessed is 8-bits wide. Hence, each register occupies 4 bytes of memory space (32 bits / 8 bits-byte = 4 bytes). Normally, the first accessible register of the component can be accessed using the component's base address defined in Qsys, and the second accessible register will be at the address which is 4 bytes after that, and so forth.

The **IORD** and **IOWR** functions provided by "**io.h**", can be used for reading and writing respectively. The IORD and IOWR functions are actually macros that use built-in assembly functions to perform memory mapped register operations. Also note that there are 3 variants for each function, resulting in a total of 6 functions: IORD_8DIRECT, IORD_16DIRECT, IORD_32DIRECT, IOWR_8DIRECT, IOWR_16DIRECT, and IOWR_32DIRECT, where the numbers (8, 16, and 32) indicate the width of the data for that particular operation. **IORD and IOWR by themselves are sufficient for performing all required operations**.
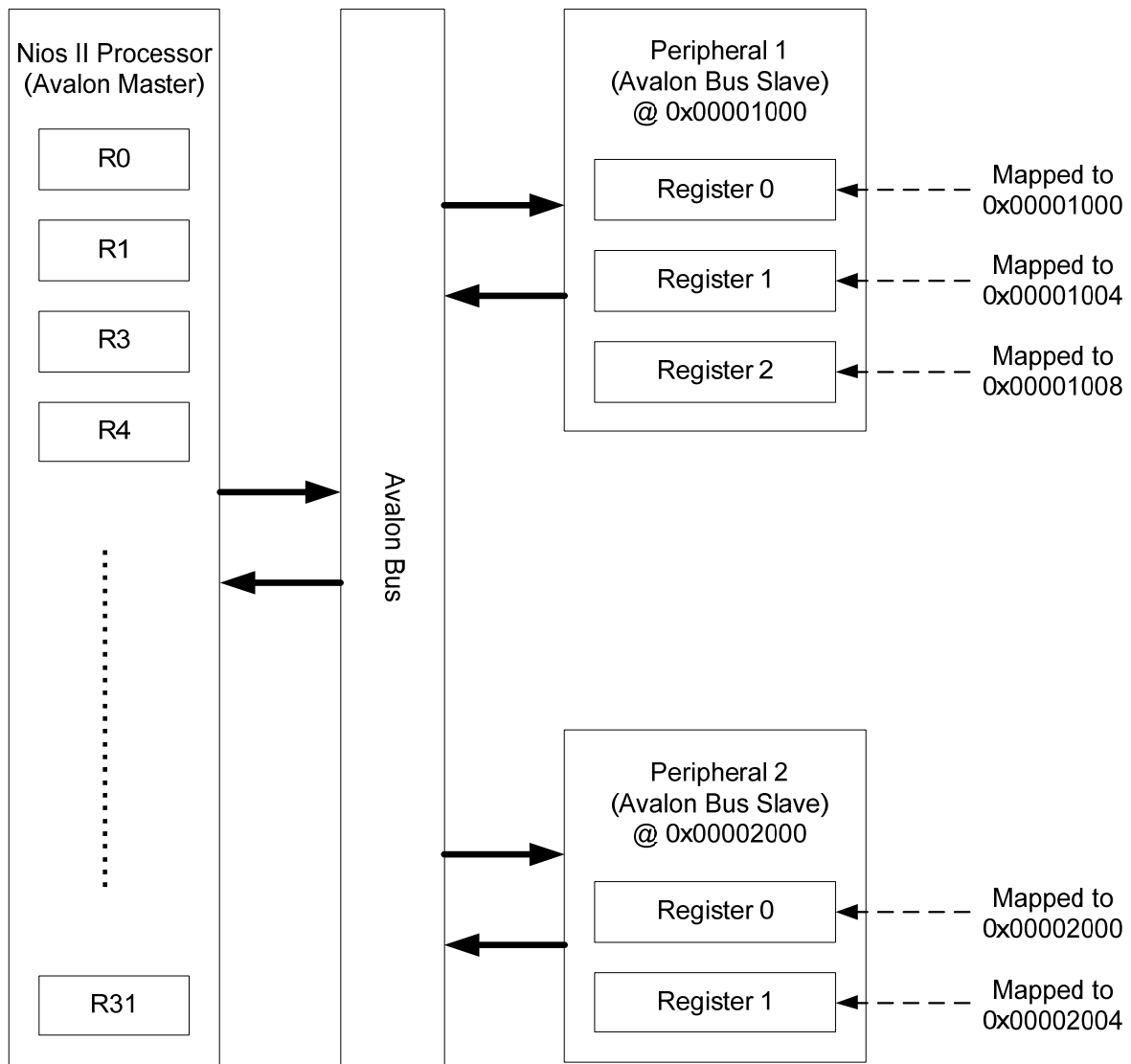
Figure 4.2: Nios II and peripherals and memory mapped I/O registers

More insight into system.h and the HAL APIs

Extensive use of IORD and IOWR is inconvenient. Application programmers should not need to remember the function of every single register in a complex system. Futhermore, code that makes extensive use of IORD and IOWR is likely to be less readable. Hence, Altera provides a set of HAL (hardware abstraction layer) APIs (application programming interface) to access memory mapped components in an easier and more sensible way.