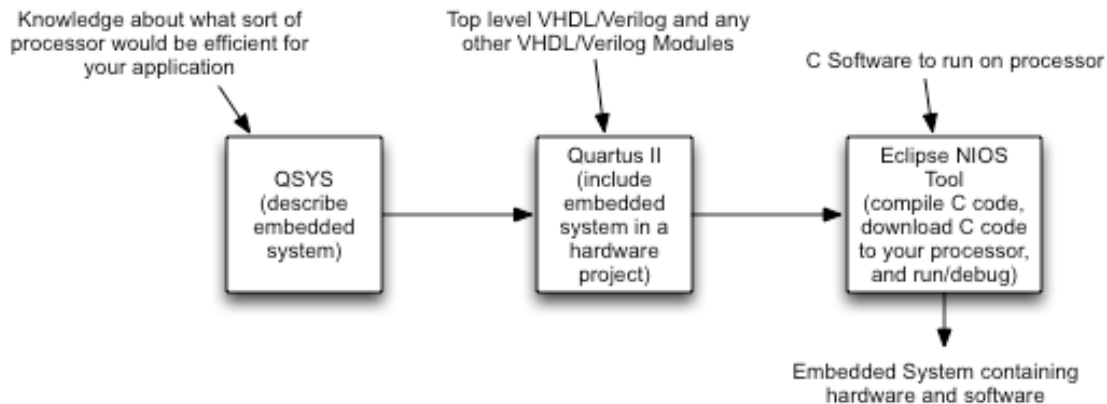


**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
2015/2016 Term 2**

**Tutorial 1.4
Using the Eclipse-Based NIOS Programming Tools**

In Tutorials 1.2 and 1.3, you compiled, downloaded, and ran your C Software using Altera Monitor Program (AMP). In this tutorial, we will replace AMP with a better, industrial strength, tool. Our design flow will then look like:



This tutorial will only discuss the last box, the use of the Eclipse NIOS tool. The other boxes in the flow are the same as in Tutorials 1.2 and 1.3.

AMP is a very basic program intended only for university students. No one in industry would use AMP. Rather than use AMP, we will use the industrial-strength tool for the following reasons:

1. The industrial tool is based on Eclipse, which is a standard program development environment used in industry today. Programmers use Eclipse to write code in C++, Java, Python, and many other languages. Eclipse will be an important skill for your resume.
2. The industrial tool has much better support for C programming (rather than assembly language programming). In 2nd year (EECE 259), you programmed an embedded processor using assembly language. This was a great way to learn the inner-workings of the processor, but today, industry rarely uses assembly language. C is much more common. Although AMP does support C, the debugging features for C are very weak.

Before completing the following steps, be sure to complete Tutorials 1.2 and 1.3. You will start with your compiled design from Tutorial 1.3 containing SDRAM.

1.0 Running Eclipse

Open your project from Tutorial 1.3 in Quartus II and choose “**Tools->NIOS II Software Build Tools for Eclipse**” from Quartus II. This will run Eclipse. The first time you run it, you will see a window similar to that in Figure 1. The workspace is a location where Eclipse will store all your files and directories. Enter a location for the workspace (in my system, I am using my Z drive, however, you probably would choose something like “**C:\workspace**” on your own PC).

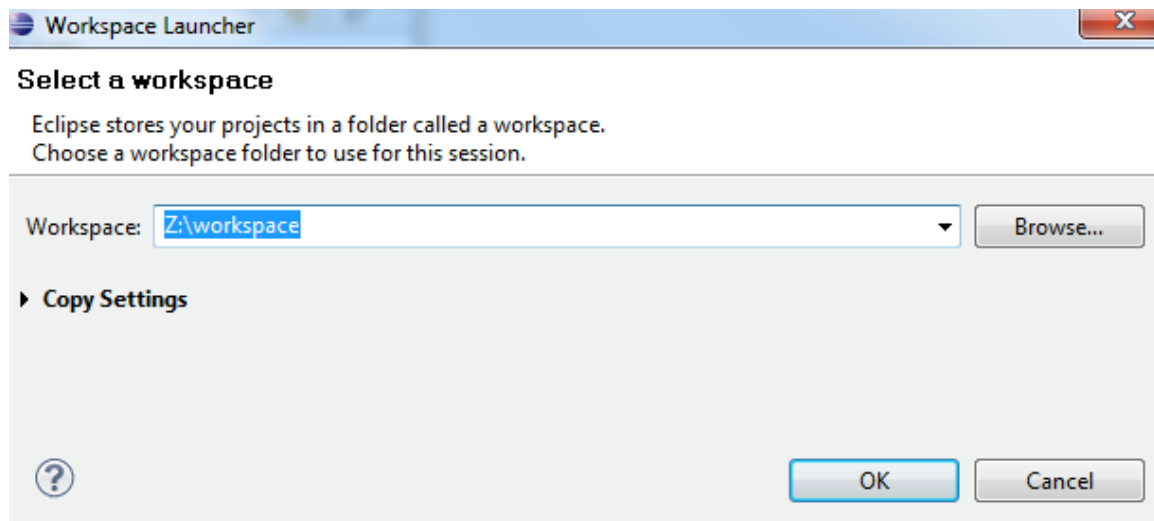


Figure 1: Setting a workspace

The main Eclipse window will then open, looking something like Figure 2. To ensure the NIOS II perspective is visible, click on the NIOS II button near the top right of the screen.

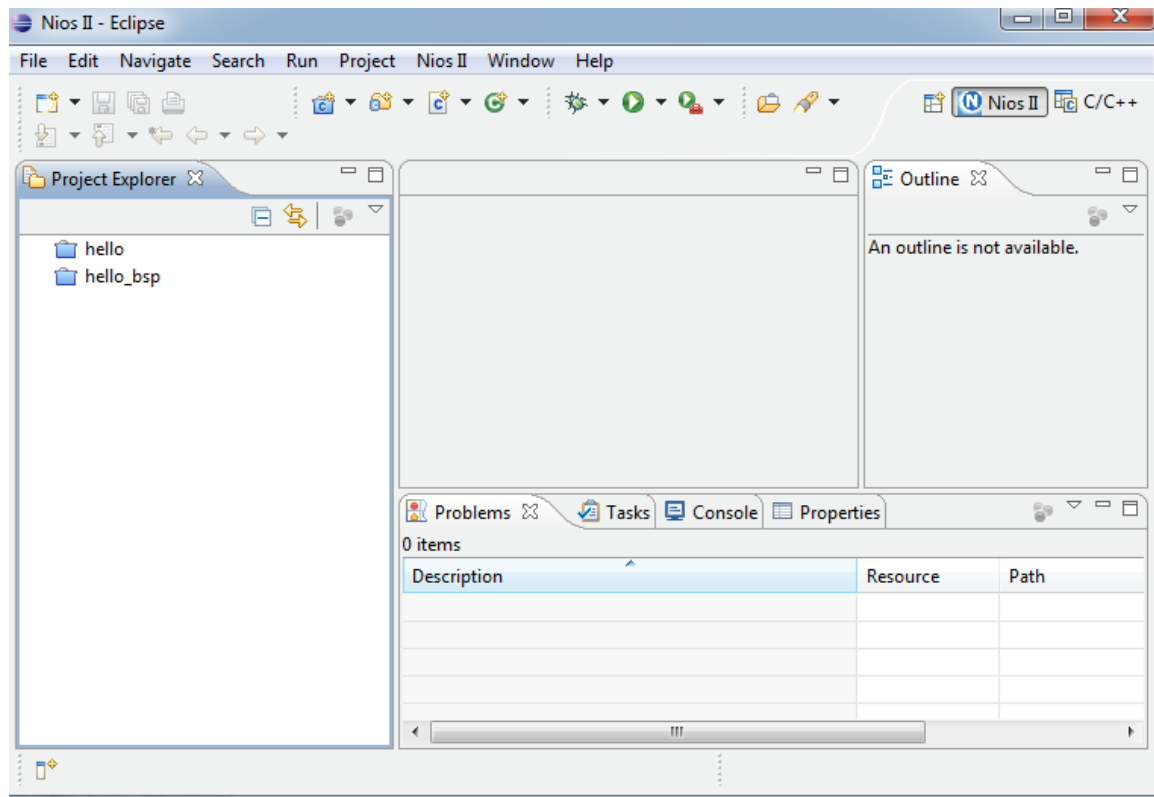


Figure 2: Eclipse Main Window

Creating a New Project:

First some terminology. A *Board Support Package (BSP)* is a package describing all the physical details of where you will run your software. This contains information about the board itself (in our case, the DE2 board) as well as the NIOS II processor you have constructed (which peripherals have been included in the processor, base addresses for each peripheral, etc). As we will see below, the software we are using is smart enough to read a file in your Quartus II project that contains this information. The C compiler is then able to use this information as it compiles your code.

A second piece of terminology: A *Project* is simply a collection of files related to a design. They may be source files, object files, or other files.

Your application will actually be contained in *two* projects. One of the projects will contain your application code (source code, etc), while the other project will contain the Board Support Package (BSP). This may seem non-intuitive at first, however, it cleanly separates the design of your software from the design of the processor upon which the software is running.

To start developing an application, you need to create your two projects. To do this, choose **File->New->NIOS II Application and BSP From Template**. This will lead to a window similar to the one in Figure 3. In this window, you must first enter the name of a file that describes the processor hardware you created in Quartus II. Click on “..” and find the file **nios_system.sopcinfo** (this file was created when you clicked “generate” in QSYS). Highlight this file, and choose Open. After a delay, you will see this file name appear in the first field under “Target hardware information”. You will also see an indication of the name of the processor in your design. In the tutorial, your processor was named “**nios2_processor**” so you should see this beside “CPU Name”. Note that if your design had more than one processor, you could select a processor from this dialog.

You then need to choose a project name. For this tutorial, use **example** (enter this string after “project name”).

The box labeled “Project Template” lists some pre-designed software modules that you can use as a starting point for your own design. This allows you to quickly start developing your code. For this tutorial, choose the **Hello World** template (which is a program that simply prints the string “Hello World” to the console).

When you are done with this, click on Finish, and the tool will create two projects. The first project is called “**example**”, which will contain all your application software, and the second project is called “**example_bsp**” which contains the Board Support Package, as described above. You will see both of these listed under “Project Explorer”.

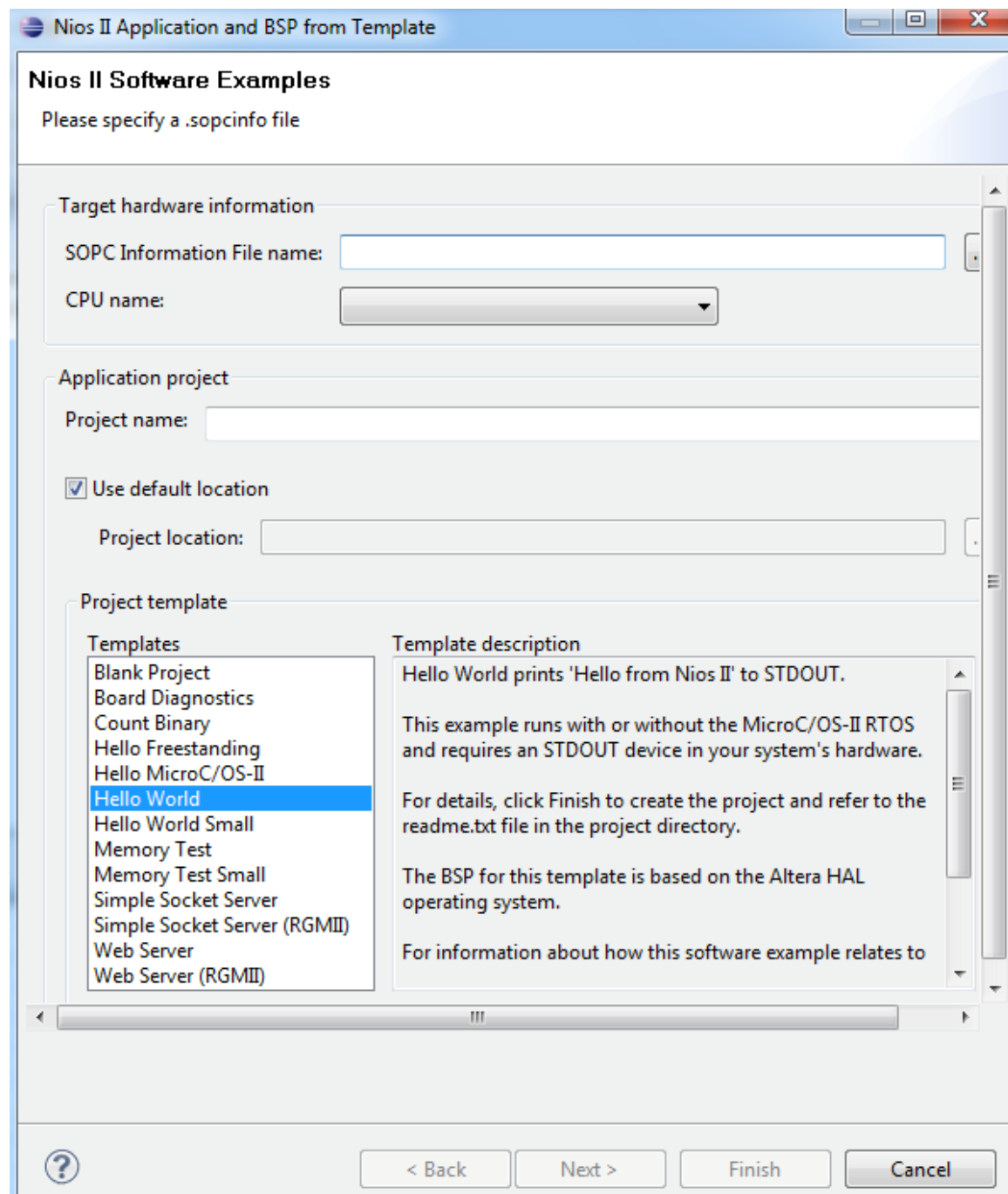


Figure 3: Creating a new Application and BSP

2.0 Generating the BSP

In the previous step, you created two projects, one of which contains your BSP. You must now “generate” the BSP from this project. You can do this by right-clicking `example_bsp` in the Project Explorer window, and choosing “**Nios II -> Generate BSP**”. A dialog box will open for a short time (a few seconds) and then close. The BSP has now been generated. You must remember to do this every time you create new hardware in Quartus II (for example, if you ever go back to change the structure of your processor such as adding new peripherals or fixing any connection patterns between the peripherals).

3.0 Writing and Compiling your Application Code.

Now, double click the “example” project to see what is there. You will see something similar to Figure 4. The most interesting file here is “**hello_world.c**” which contains the `main()` function of your application. Double click `hello_world.c`, and a file similar to that in Figure 5 will open.

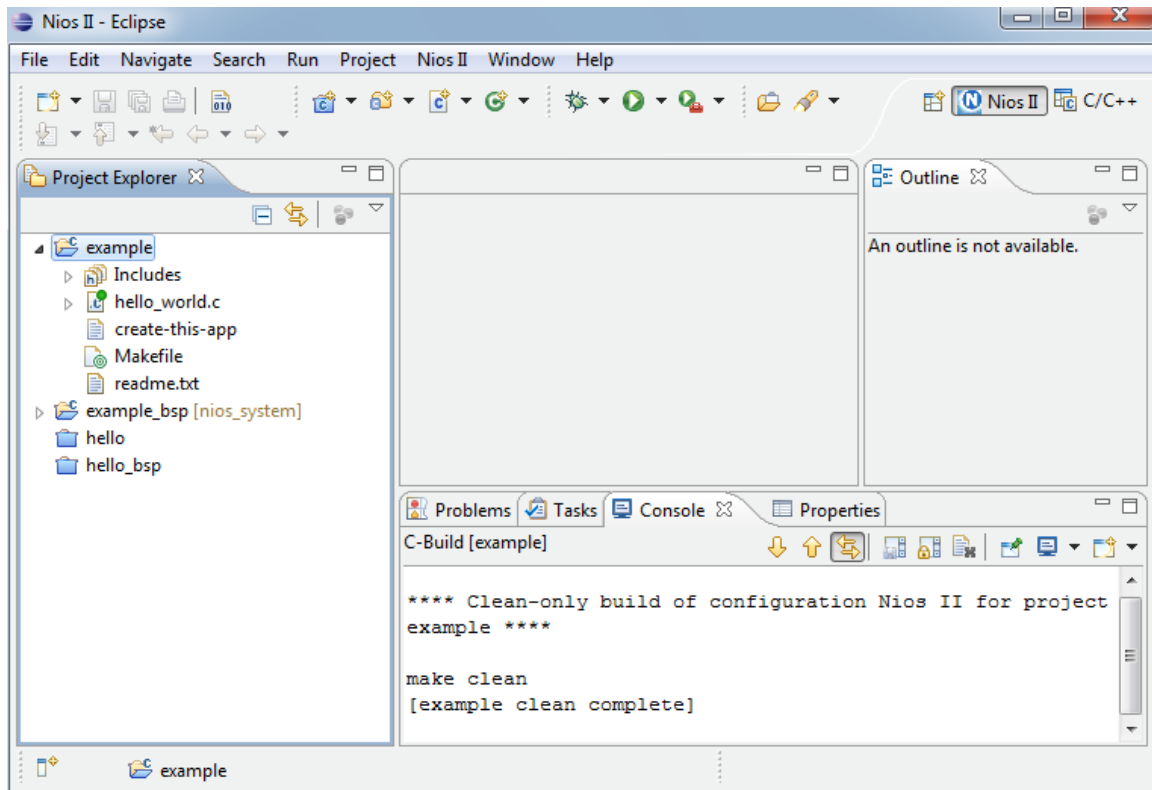
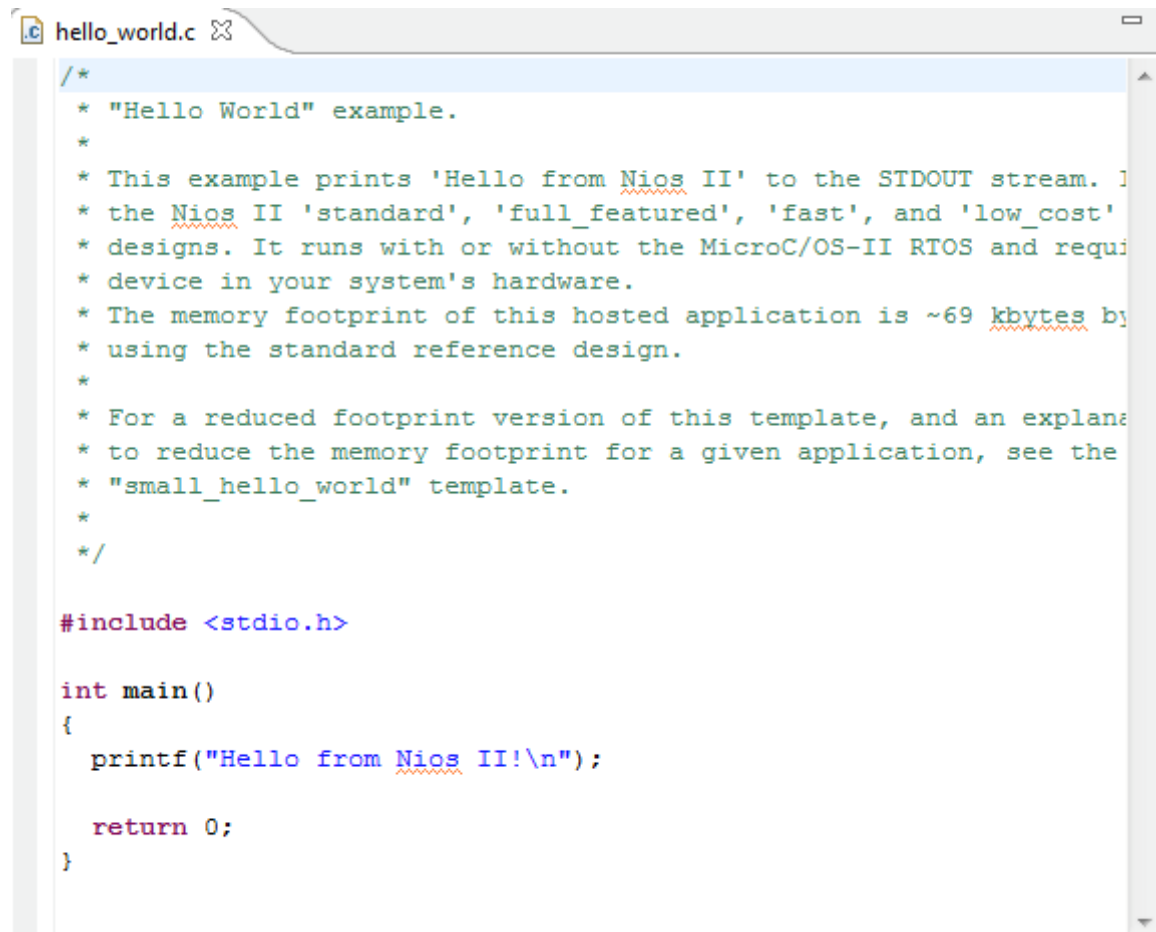


Figure 4: Opening up the "example" project



```
hello_world.c
/*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It
 * uses the Nios II 'standard', 'full_featured', 'fast', and 'low_cost'
 * designs. It runs with or without the MicroC/OS-II RTOS and requires
 * a device in your system's hardware.
 * The memory footprint of this hosted application is ~69 kbytes by
 * using the standard reference design.
 *
 * For a reduced footprint version of this template, and an explanation
 * to reduce the memory footprint for a given application, see the
 * "small_hello_world" template.
 */

#include <stdio.h>

int main()
{
    printf("Hello from Nios II!\n");

    return 0;
}
```

Figure 5: Hello_world.c template application

Just to prove you can, change the string to something more interesting (like “EECE 381 is challenging, but I’m going to learn so much!\n”). Don’t forget the “\n” at the end; this is the “new-line” character. Save the file.

Programming the FPGA:

Before running your software, you must program the FPGA with your NIOS processor. I normally would program it within Quartus II immediately after compiling (using the same techniques you learned in EECE 353), however, you can do it from within Eclipse too. To do this, from the top menu, choose **NIOS II->Quartus II Programmer**. This will open the familiar window in Figure 6. Be sure that the USB-Blaster is specified as the hardware (if not, choose Hardware Setup” to select it; you still can’t find it, make sure your board is plugged in properly).

You must also choose your programming file. Choose “**Add File**”, and select the **.sof** corresponding to your design, as in Figure 7. You may have to navigate to your Quartus II project directory. Note that, in this example, although the Quartus II project was named “**lights**”, the programming file is named

“lights_time_limited.sof”. To understand why, you need to know a bit about how Altera makes money. Most of Altera’s income comes from selling parts. However, they also sell IP cores. Some IP cores supplied by Altera are not free. However, Altera provides a free “evaluation method” for using these cores. Systems containing these cores can be run as long as the DE2 board is connected to the host using the JTAG cable (the cable you use to download your design). If your design contains any of these cores, the design is said to be “time limited”, and the sof file will contain the suffix **time_limited** as in our example. The intention is that you can design a system around these cores for free, but as soon as you go into production, you need to pay Altera for using the cores. There are some other implications of using these cores, which will be explained soon.

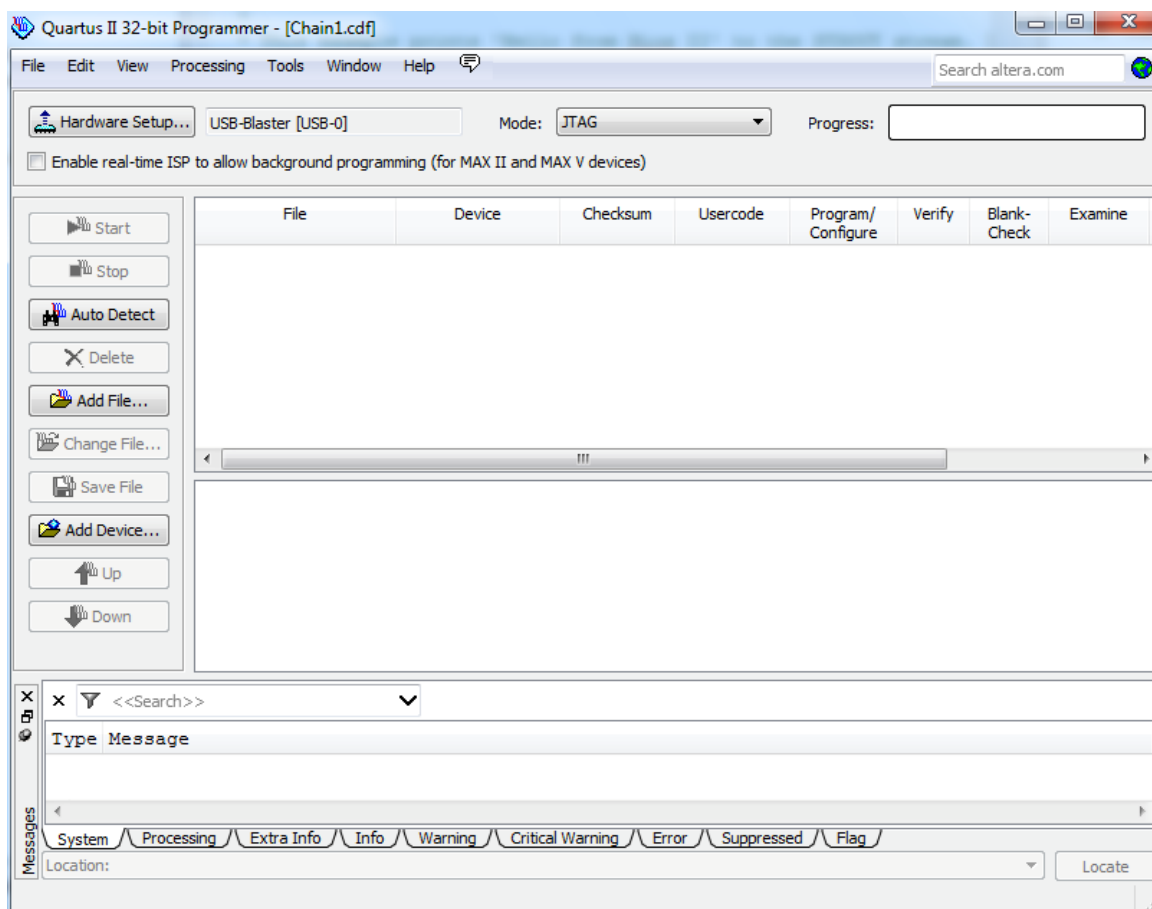


Figure 6: FPGA Programmer

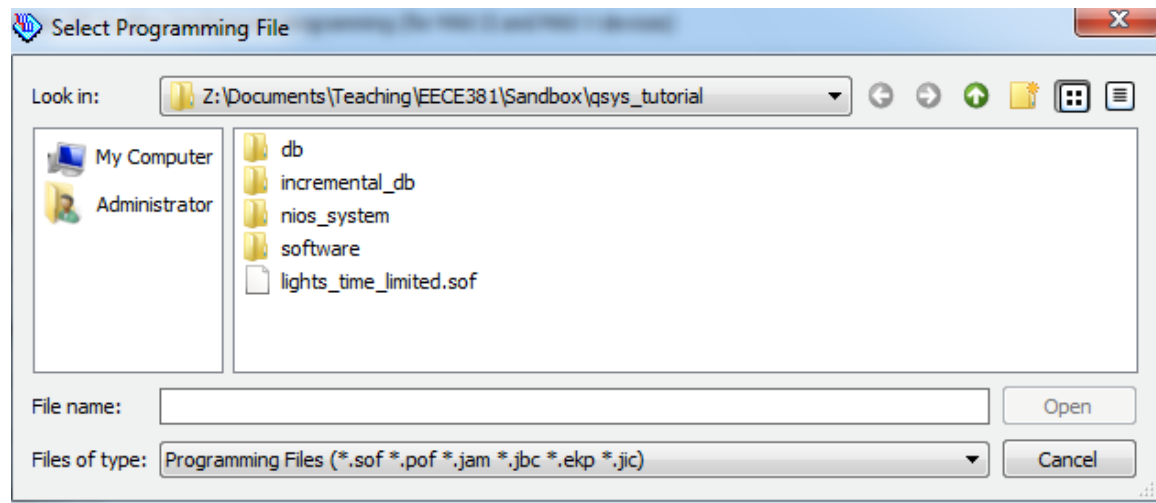


Figure 7: Select programming file

Highlight “**lights_time_limited.sof**” and click Open. You will be confronted with a window that looks something like Figure 8. Click OK.

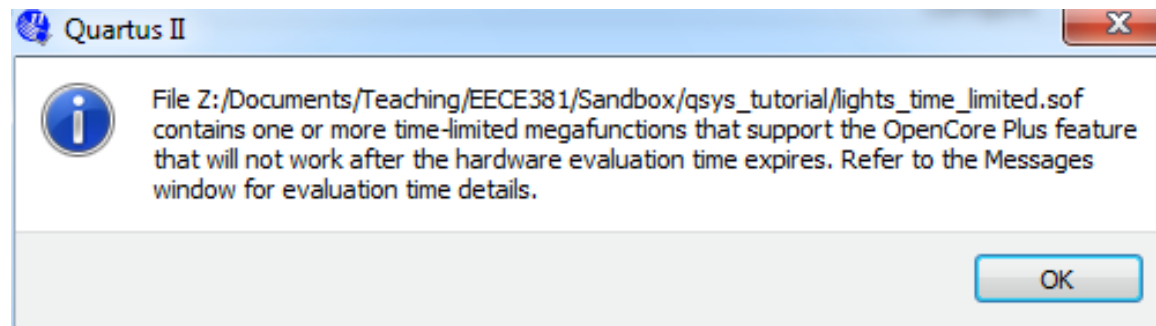


Figure 8: Warning message that tells you your design contains time-limited cores

You should then be able to click Start on the programming window. If successful, your NIOS II will be downloaded to the board, and you will see a window similar to Figure 9. *Do not close this window!* Your NIOS II system will only work if this window is open, and if the DE2 board is connected to your PC through the JTAG cable. If you close your window, or unplug the board, the NIOS II will stop.

Once you have finished this step, the FPGA will be configured as your NIOS II processor. However, since the processor is not running any software, it won't appear to do anything. The next step is to download the software to the processor and run the software.

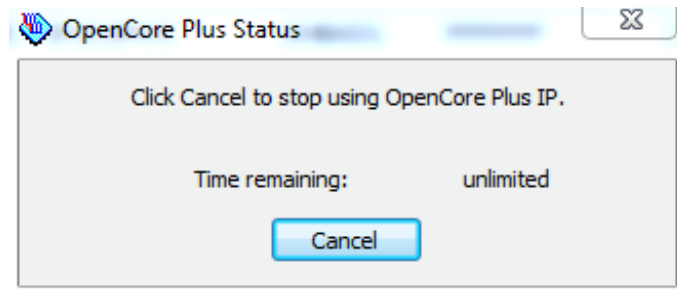


Figure 9: Do not close this window!

3.0 Running your application:

Now, right-click "**example**" under the Project Explorer. Choose "**Run As -> NIOS II Hardware**". Your design will be compiled (this will take some time, especially the first time you run it, since all the support routines must also be compiled). During the compilation you will see messages in the "console" pane at the bottom of the screen. This is the communication portal between you and Eclipse. Eventually, you will see it download your software to your processor. It will then switch the view of the bottom pane from "Console" (which is used for messages from Eclipse) to "NIOS-II Console" which is used for messages to and from the NIOS processor on your board). It will then run the software.

You may get a message telling you that there are errors in your C program and asking if you want to continue. It doesn't make sense to continue if there are errors in your C program, so click **No**, and go back to your **hello_world.c** and make sure it is correct. A common error is to forget semicolons.

If all this works correctly, you should see a message from the NIOS processor that looks something like Figure 10. If so, congratulations, you have successfully compiled and run your program.

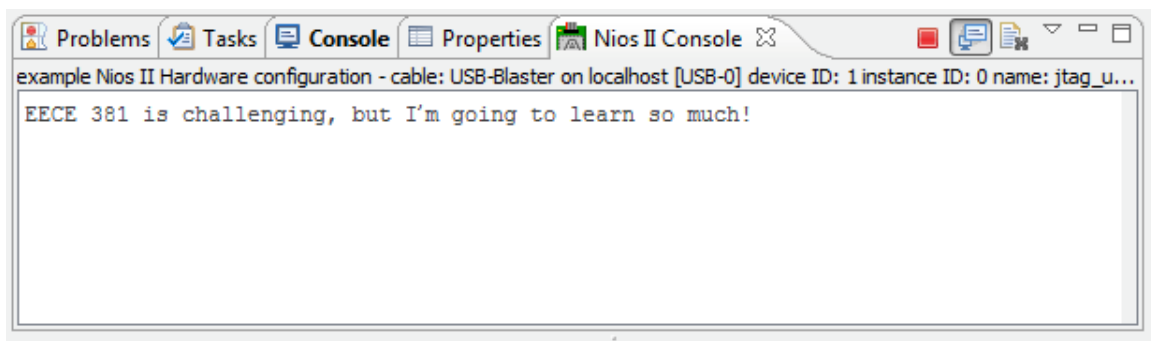


Figure 10: Output of your program running on the NIOS processor

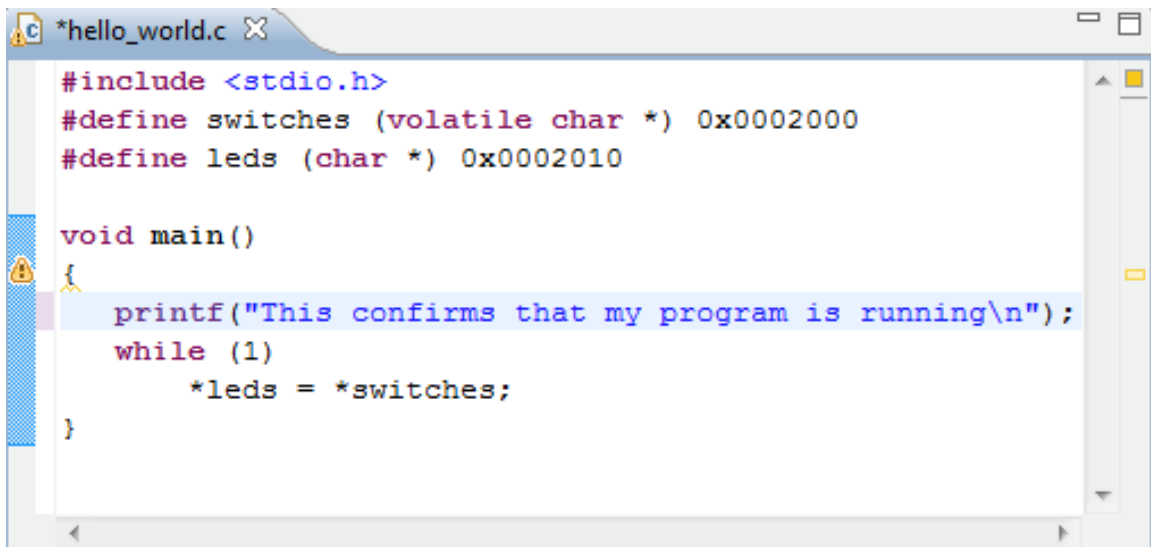
4.0 Communicating with Peripherals:

In the previous example, our program communicated with the outside world by writing to the console. In most embedded applications, we want to do more than that. We usually want to communicate with the various peripheral devices that we might have added when we built our processor using QSYS. As an example, in the program you wrote in Tutorial 1, you first read the switch positions through an input parallel port and then drove the lights through an output parallel port.

There are two ways to communicate with peripherals: low-level “bare metal” programming, and through the use of hardware drivers. We will explain both. Although the use of hardware drivers leads to simpler code, and higher development productivity, “real engineers” often enjoy programming at the lowest level possible. It is also important to know understand “bare metal” programming so that you can debug when things go wrong with hardware drivers.

Bare-Metal Programming:

In this simple example, we will perform the same task as the program you wrote in the first QSYS tutorial. Expand your project by clicking on “**example**” in your Project Explorer, and then double click on **hello_world.c**. Change the code to look like Figure 11 (this is the same as the code you wrote in that tutorial, although I’ve added a **printf** statement just so you can confirm from the console that your program is running). As in that tutorial, make sure the addresses of the two parallel ports match those addresses specified when you developed your processor using QSYS.



```
*hello_world.c

#include <stdio.h>
#define switches (volatile char *) 0x0002000
#define leds (char *) 0x0002010

void main()
{
    printf("This confirms that my program is running\n");
    while (1)
        *leds = *switches;
}
```

Figure 11: Example program showing “bare metal” programming

Run this program as before (right-click on example and then choose **Run As -> NIOS II Hardware**). The program will compile, it will be downloaded, and you will see the message in the console. Change the switches on your board, and you should see the lights change, as in the first tutorial.

The reason this is called “bare metal” programming is that you are talking to the hardware (by reading and writing directly from/to specific memory locations) rather than through a software layer. The advantage is flexibility; you can read or write anything to any memory location allowing you to drive your peripherals with whatever you want (we will talk more about memory mapping later). The downside is that you need to hardcode your memory locations (eg. in the above code, the switches are hardcoded to location 0x002000). If you changed your QSYS design, and the location changed, your code would no longer work. The other downside is that it would be difficult for someone else to understand your code. In industry, you will rarely be allowed to use “bare metal” programming (especially in a product), however, every good engineer should know how to do it.

Hardware Abstraction Layer:

The other option is to employ a hardware abstraction layer (HAL). This is a software layer that abstracts the low level memory reads and writes, and provides a somewhat simpler API to the programmer. Each core in the Altera system comes with a set of function calls that make up a hardware abstraction layer. By limiting all interactions between your code and peripherals to go through one of these functions, your code will be more portable and easier to understand.

Unfortunately, the Parallel Interface peripheral you are using is so simple, it does not contain such a set of functions. In a later tutorial , we will integrate a much more complex peripheral, and will illustrate the use of these functions at that time.

Solutions to Common Problems:

Below are solutions to common problems related to Eclipse that we have encountered in the past. Other tips will be provided in the labs or on the Connect bulletin board as we encounter them. For all of these, the TAs can help you if you have problems, but this list will give you a place to start.

1. If you receive a pop-up error when they try to "create a new BSP and application from template" that reports that it can not execute a makefile, try the following:

Create the BSP and the Application Folder separately: **File->New->NIOS II Board Support Package** (give it a name something like example_bsp)
then: **File->New->Nios II Application** and point it to the BSP you just created.

2. Last year, two people have seen an error related to not finding "_main" when they compile their Eclipse project. This is due to a bug in which the top-level C file is not included in the **makefile**. In the unlikely event that you encounter this, create a new project, and add a new top-level file yourself (right click on the project name, and add new file). This will create a new top-level file and will add it to the **makefile** for you.

3. If you have problems downloading your design and running it on the board, try the following troubleshooting checklist.

- A. Ensure that your program compiled without errors. (Make sure there isn't a red X next to the project name)
- B. Refresh the connection to the board. Right-click your project, **Run As->Run Configurations...** On the 'Target Connection' tab: 1) Check 'Ignore mismatched system ID ' 2) Check 'Ignore mismatched system timestamp', and 3) click 'Refresh Connections'.
- C. Ensure that your top-level HDL file is correct. The character LCD, VGA etc (which will be introduced in later exercises/tutorials) all require extra connections in your top-level HDL file. You need to make sure you are using the correct top-level I/O names, and that your component definition of your processor system is correct. To check that your signal names match the pin assignments: Open the Pin Planner (**Assignments->Pin Planner**), and filter by 'Pins: Unassigned'. The list should be empty. If there is anything in this list, it means it is a top-level signal that is not properly connected to a pin. To check that your top-level component definition is correct: in Qsys look at the 'HDL Example' tab (select VHDL as your language). The component definition here should match what is in your top-level HDL file.
- D. Double check your reset connections. Every core should have a reset signal from the **jtag_debug_module_reset** and the global **clk_reset**.
- E. Make sure you have imported the pin assignments in Quartus II (easy to forget).
- F. Double check the settings of the DRAM core