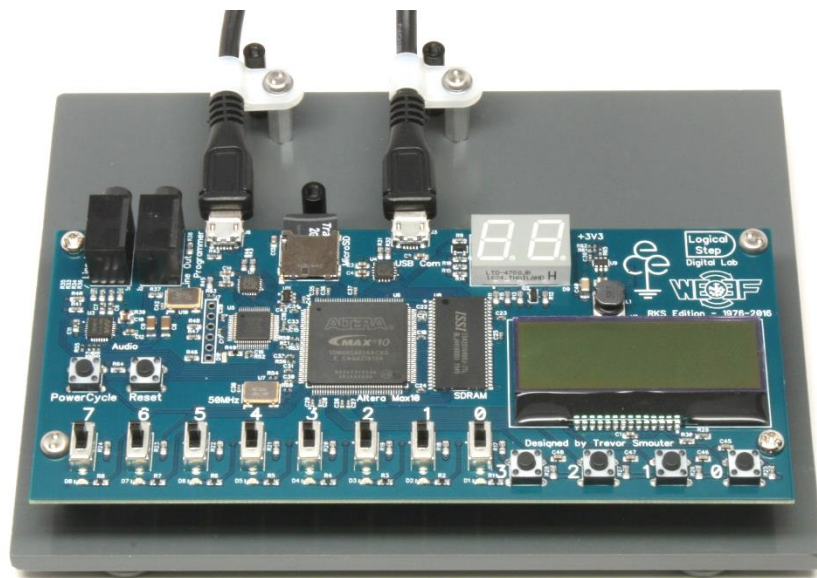


University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering



ECE-224 F2017
Lab Manual for Embedded Microprocessor
Systems – v3.3

Using this Lab Manual Effectively

Considerable care has gone into developing this lab manual. The philosophy behind its creation was to provide a document that complements in-lab instruction and support. This learning aid is not intended to stand on its own as a complete resource which would allow students to complete the labs successfully on their own – it is designed to complement a lesson plan delivered in the lab. This document also intentionally avoids incomplete coverage of details that are well documented elsewhere. If required, you will be provided with advice on where to get the information you require.

Throughout this document there are textboxes that highlight different types of information to facilitate learning. The purpose of the various textboxes is described below.



Watch out! —

The Watch Out textbox is used to provide cautionary information that can help you avoid common pitfalls that students experience. These pitfalls can result in road blocks that are either time consuming to fix or will possibly require the help of an expert such as the Lab Instructor or TA's to solve.



Deep Dive —

The Deep Dive textboxes are used to provide background information about the topic and usually contain advanced level information. If you're interested in getting more familiar with the topic or want to understand at a deeper level, then read these textboxes. If you are a beginner struggling with the concepts introduced, you can safely ignore the information in these textboxes and move on with the core concepts.



Tips and Tricks —

The Tips and Tricks textboxes provide suggestions that can help to speed up your work and generally make your life easier.



Take Away —

The Take Away textbox provides a summary of important points recently discussed in the manual. It's a good idea to read the Take Away textboxes to ensure you've gotten the main points recently introduced and haven't missed any details.

Let's try out some textboxes to see them in action.



Tips and Tricks

In this document when you see underlined and blue words presented like the following, 'See: [Tips on writing lab reports](#)' it is a recommended search engine phrase to find out more details about the topic online.



Watch out!

The software tools introduced in this lab are complex and targeted at an expert level audience (as is the case with all FPGA development tools). Tool problems that impede development are common. Ensure that you use your scheduled lab time wisely - every hour spent in the lab is equivalent to three hours working without the support of the Lab Instructor and/or TAs.

Acknowledgements

A number of people have contributed to the creation of this document and have also provided valuable input to the evolutionary content. We express our gratitude to the following people for their contributions to the Lab Manual:

Trevor Smouter, MASC., P.Eng.: Creator of the original document and designer of the LogicalStep board;
Charles K. Pope, BASC., P.Eng: Lab Instructor;
Roger K. Sanderson, BASC., P.Eng.: Lab Instructor and co-creator of the original labs material;
Bill Bishop, PhD., P.Eng.: Course Lecturer;
Allyson Giannikouris, MASC., P.Eng.: Course Lecturer;
Rodolfo Pellizzoni: Course Lecturer;

Table of Contents

List of Figures	vi
List of Tables	viii
1 Introduction	1
1.1 Altera Toolchain	1
2 Lab 0: Developing the Lab Design Platform	4
1.2 Lab 0 Hardware Development.....	4
2.1.1 The Top File	7
2.1.2 Assigning Pins	8
2.1.3 Qsys	8
Adding the 'Avalon ALTPLL' IP Core	12
Adding the 'NIOS II Processor' Module	14
Adding the 'SDRAM Controller' IP Core	14
Adding the 'System ID Peripheral' IP Core	15
Adding the 'JTAG UART' IP Core	15
Making Qsys Connections	16
Adding the 'PIO' IP Core for LEDs	18
Adding the 'PIO' IP Core for Push Buttons	18
Adding the 'PIO' IP Core for Switches	19
Adding the 'Altera Avalon LCD 16207' IP Core.....	19
Adding the 'Audio and Video Config' IP Core.....	20
Adding the 'Audio' IP Core	20
Adding the 'UART (RS-232 Serial Port)' IP Core.....	21
Adding the 'Interval Timer' IP Core	22
Adding another 'Interval Timer' IP Core.....	22
Adding the 'SPI Master (3 Wire Serial)' IP Core.....	22
Adding the 'Dual 7 Segment' IP Core	22

Adding the 'EGM' IP Core	23
Adding the 'PIO' IP Core for Stimulus_in.....	23
Adding the 'PIO' IP Core for Response_out.....	24
2.1.4 Editing the Top File to Instantiate the Qsys Project	26
2.1.5 Compiling the Project	27
2.1.6 Hardware Wrap-up	27
2.2 Lab 0 Test Software Development	28
2.3.1 Board Diagnostics Project in the NIOS II Software Build Tools	28
2.4 Lab 0: Demo.....	33
2.5 Lab 0 Post-Lab Exercises: Getting to know the NIOS/Eclipse IDE	33
2.7.1 Lab 0 Post-Lab Exercises: Setting up a Simple Read/Write Operations Project	36
2.7.2 Lab 0 Post-Lab Exercises: Creating a Simple Read / Write Looping Routine	42
3 Lab 1: Experimenting with Polling and Interrupts	43
3.1 Introduction.....	43
3.2 Overview.....	43
3.3 Lab 1 Software Project Setup	44
3.4 EGM Module and its Use in Lab 1	46
3.5 Lab 1 Code Development for Executing a Test.....	48
3.5.1 Recommended Approach to Interrupt Test Development	49
3.5.2 A Possible but Incorrect Approach to Tight Polling Test Development	49
3.5.3 Recommended Approach to Tight Polling Test Development	50
3.6 Lab1 Code Development for Executing the Experiments.....	52
3.7 Lab 1 Objectives and Deliverables.....	53
3.7.1 Lab 1 Demo	53
3.7.2 Lab 1 Experimental Results for the Lab1 Report	54
3.7.3 Lab 1 Report	54
4 Lab 2 Based Practicum: Engineering Design Services Case Study	56
4.1 A Message from the LogicalStep Company's Solution Architect	57

4.2	Client Provided Design Specifications for Audio Player Product.....	60
4.3	Timesheet.....	61
4.4	Lab 2 Objectives and Deliverables.....	61
4.4.1	Prelab Timesheet Exercise	62
4.4.2	Demonstration	62
4.4.3	Reflection Report	62
Appendix A: Custom IP Cores		63
Appendix B: NIOS Interrupts		65
Appendix C: Solving NIOS Software Download Issues		71
Appendix D: Lab 0 Activity H/W Development Checklist		74
Appendix E: Lab1 Logic Analyzer (SignalTap).....		76
Appendix F: Lab2 Hardware Testing		81
Appendix G: Lab2 FatFS Commands		82
Appendix H: Lab2 Logic Analyzer (SignalTap)		84

List of Figures

Figure 1. Quartus Prime and the Nios II Eclipse tool relationship.....	2
Figure 2: Launching the Altera Quartus 15.1 tools.....	4
Figure 3: New Project Wizard.....	5
Figure 4: The 'Add Files' dialog box.....	6
Figure 5: Family and Device Settings dialog box.....	6
Figure 6: Selecting the top file in the Project Navigator.....	7
Figure 7: Hierarchy for the LogicalStep design.....	7
Figure 8: Clock Source Core Parameter Settings.....	10
Figure 9: Naming an IP Core in Qsys.....	10
Figure 10: Export Settings of a Core in Qsys.....	10
Figure 11: AltPll Core Parameter Settings.....	12
Figure 12: AltPll C0 Parameter Settings.....	12
Figure 13: AltPll C1 Parameter Settings.....	13
Figure 14: AltPll C2 Parameter Settings.....	13
Figure 15: Removing the AltPll locked output.....	13
Figure 16: SDRAM Controller Core Parameter Settings.....	14
Figure 17: First Connections in Qsys.....	16
Figure 18: LED PIO Core Parameter Settings.....	18
Figure 19: Push Button PIO Core Parameter Settings.....	18
Figure 20: Switch PIO Core Parameter Settings.....	19
Figure 21: Audio and Video Config Core Parameter Settings.....	20
Figure 22: More Qsys Core Connections.....	21
Figure 23: Stimulus in PIO Core Parameter Settings.....	23
Figure 24: Response out PIO Core Parameter Settings.....	24
Figure 25: Remaining Qsys Connections in Qsys Window.....	24
Figure 26: NIOS Memory Vector Parameters.....	25
Figure 27: Qsys Project Generation.....	25
Figure 28: Quartus Programmer Setup with USB cable connected.....	27
Figure 29: Launching the Altera NIOS II Software Build Tools.....	28
Figure 30: Eclipse Workspace Setup.....	29
Figure 31: NIOS II- Eclipse Window.....	30

Figure 32: NIOS II Application and BSP from Template option	31
Figure 33: NIOS II Project Creation.....	32
Figure 34: Referencing the system.h info from the BSP Project	34
Figure 35: Examining the Button_PIO_Base Address in the system.h info	35
Figure 36: Looking at the Board_Diagnostics Source Code.....	36
Figure 37: Configuring a New Application from a Template for Lab_0_Post_Lab	37
Figure 38: Creating a BSP Project to couple with the Lab_0_Post_Lab Project.....	38
Figure 39: New Lab_0_Post_Lab Project is OPEN	39
Figure 40: Accessing the Lab 0_Post_Lab "Hello_World" c-code	39
Figure 41: Segment Mapping for Dual 7 Segment Core.....	41
Figure 42: Configuring a New Application from a Template for Lab_1.....	44
Figure 43: Changing the sys_clk_timer Settings in the BSP.....	45
Figure 44: Enabling the Small Driver for the JTAG UART	46
Figure 45: EGM and NIOS Topology	46
Figure 46: EGM Stimulus Pulse-train Settings	47
Figure 47: EGM Latency Measurement.....	48
Figure 48: Running a “given number” of Background Tasks during Tight Polling.....	50
Figure 49: Characterizing Background Tasks during Tight Polling.....	51
Figure 50: Segment Mapping for Dual 7 Segment Core.....	63
Figure 51: EGM stimulus pulse	64
Figure 52: Interrupt Connections in Qsys.....	66
Figure 53: Project Directory with SignalTap File for Lab1	76
Figure 54: SignalTap Console for Lab1.stp	77
Figure 55: Inclusion of the Lab1.stp file in your LogicalStep Project	78
Figure 56: Enabling the SignalTap Logic Analyzer for the FPGA.....	79
Figure 57: SignalTap Capture	80
Figure 58: SignalTap Console for Lab2.stp	84

List of Tables

Table 1: Clock Source Core Name and Exports	10
Table 2: AltPll Core Name and Exports.....	14
Table 3: NIOS II Core Name and Exports	14
Table 4: SDRAM Controller Core Name and Exports.....	15
Table 5: System ID Peripheral Core Name and Exports	15
Table 6: JTAG UART Core Name and Exports	15
Table 7: LED PIO Core Name and Exports	18
Table 8: Push Button PIO Core Name and Exports.....	19
Table 9: Switch PIO Core Name and Exports.....	19
Table 10: Altera Avalon LCD 16207 Core Name and Exports	19
Table 11: Audio and Video Config Core Name and Exports	20
Table 12: Audio Core Name and Exports.....	20
Table 13: UART Core Name and Exports	21
Table 14: System Timer Core Name and Exports	22
Table 15: SPI Master Core Name and Exports.....	22
Table 16: Dual 7 Segment Core Name and Exports.....	22
Table 17: EGM Core Name and Exports	23
Table 18: Stimulus in PIO Core Name and Exports.....	23
Table 19: Response out Core Name and Exports	24
Table 20: Register Offsets for Dual 7 Segment Core	41
Table 21: Register Offsets for Dual 7 Segment Core	63
Table 22: Register Offsets for EGM Core.....	64

1 Introduction

The state of the art in FPGAs provides the hardware description language (HDL) designer the ability to define whole processor systems within the logical fabric contained within an FPGA. This powerful capability provides students with the ability to study both processors and the code that runs on them in an environment where the student can actually design and compile a new processor if required.

Here in the embedded microprocessor system lab we will be doing the labs on an Altera (Intel¹) FPGA called the Max10. This silicon device contains logic elements that can be assembled into complex arrangements such as a microprocessor simply by defining the device in a hardware description language (HDL) and downloading the compiled result into the FPGA. Altera provides a set of tools (a tool chain) to design, program, compile and download both the hardware and software descriptions into the FPGA. Altera also provides intellectual property (IP cores) which defines commonly used hardware that designers can benefit from using. In this lab we will be assembling a few of the Altera IP cores and some custom developed cores into a hardware project that will define our processor system on the FPGA. The central IP core we will be using is Altera's NIOS II (pronounced nee-os) soft-core processor and we will attach to it the peripherals we require for the lab activities. We'll get into this more a little later but for now let's discuss the Altera tool chain.

1.1 Altera Toolchain

The tool chain provided by Altera can be divided into two main tools described below:

- Quartus Prime² - This software tool is used to develop the hardware (the logic) that will be put on the FPGA. In Quartus Prime there are numerous tools which help design, synthesize, assemble, and program the HDL into downloadable logic. An important tool you'll be using in Quartus Prime is Qsys which provides designers the ability to assemble many different IP cores such as the NIOS II processor and selected peripherals into a single instance which can be placed in your design. This means all the details such as busses and bus arbitration are handled behind the scenes and you only need worry about the input and output ports to the instance compiled by Qsys.

¹ Intel has now purchased Altera. You may find online resources that have been re-branded as Intel. The software in the lab is branded Altera, so we will refer to it as such in this manual.

² We will be using Quartus 15.1 which is now known as Quartus Prime. For brevity, we often refer to it simply as Quartus.

- NIOS II Software Build Tools – This software tool is provided by Altera and built upon a popular software development environment called Eclipse. In this tool you will write the code for your custom processor. It also provides the ability to download and debug your code on the NIOS II processor.

These are the two main tools you will use in this lab and we will go into detail on how to use them in Section 2. For now, we will start with a top down view of how the tools fit together as misunderstandings here tend to cause students a lot of trouble. A quick overview should give you a better understanding of what is happening in the tools and why certain steps are required.

Figure 1 below gives a top down view of how Quartus Prime and NIOS II – Eclipse relate. In the design flow, the NIOS II processor and other IP Cores are assembled in Qsys. When the Qsys project is ready, generating the HDL synthesises the high level design and creates numerous files that serve to describe the hardware. The main Qsys –generated files we are concerned with are the *.qip, *.sopcinfo and the *_inst.v files.

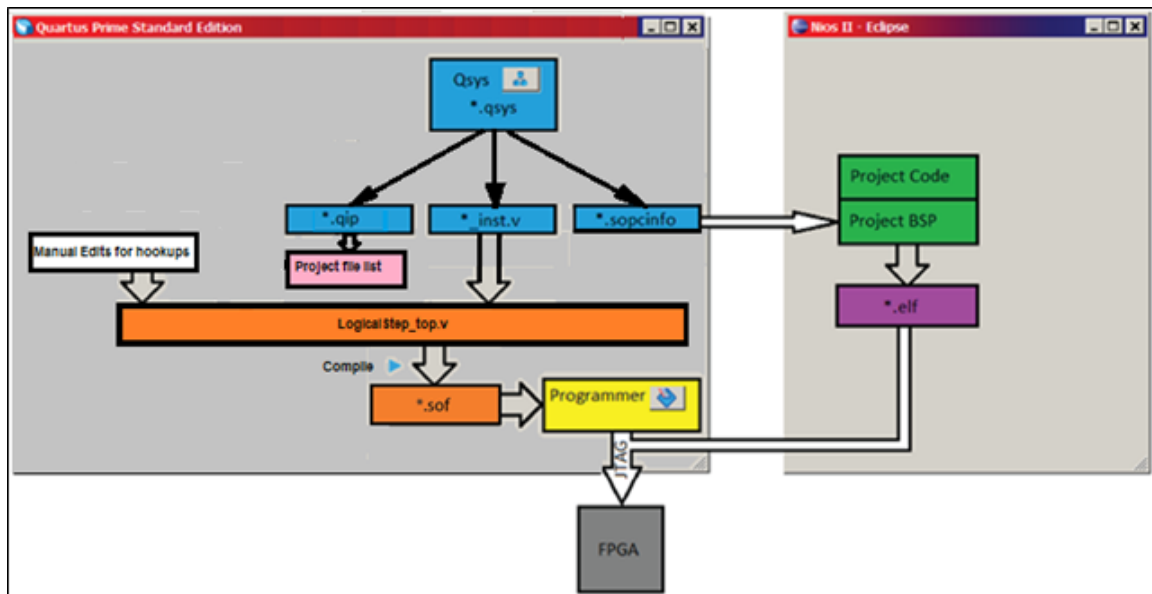


Figure 1. Quartus Prime and the Nios II Eclipse tool relationship

- 1) *.qip file: used by the Quartus tool to define the Qsys system hierarchy
- 2) *.sopcinfo file: used by the NIOS II - Eclipse tool to define the processor memory map and other details for NIOS II processor software generation and linking
- 3) *.inst file: must be instantiated by you to connect the top file signals into the Qsys module

In Quartus you define the hardware “envelope” in what is called the LogicalStep top file (this file can be thought of as analogous to a main () function in C programming). The top file also

defines the ports of the design (signals coming into or out of the FPGA) and since it is the top file these ports are special – they are the signals that are routed to the pins of the FPGA chip. They are reserved and must not be modified in any way. To this top file you will include the FPGA hardware design for the labs.

When the design entry is completed the Quartus compiler can be launched and it will start at the top file and then proceed to synthesize any logic that has been instantiated in the top file.

During the Hardware development part of this lab, you will be given the top file with the ports already declared. You will be required to develop and then instantiate the Qsys module (which will include a NIOS II processor) in the top file using the output from Qsys (*_inst.v) activity. When your top file is completed (by connecting the nets to the Qsys module), you will have Quartus compile the complete hardware design and if successful it will generate a *.sof file in the “output_files” subfolder of your project directory. The *.sof is the FPGA configuration file that gets downloaded into the FPGA via the Quartus programmer to establish the synthesized hardware design in the FPGA. When you download the *.sof file the programmer uses a JTAG connection to transfer the compiled hardware design into the FPGA. After the hardware is configured in the FPGA this connection is also employed by the NIOS II – Eclipse tool to transfer the software code into the processor as shown in Figure 1.

During the Software development part of this labs you will first create the NIOS II project workspace. Then for each project inside that workspace there will be two associated parts - the project itself and the Board Support Package (BSP). The project will contain your software program executable load file (.elf) that your NIOS II processor will run. The associated BSP is generated to create linkage between the NIOS II software and the FPGA hardware. It does this by using information from the *.sopcinfo file mentioned earlier to link all the hardware addresses etc. to the NIOS II processor program.

Upon successful compilation of the NIOS II project an *.elf file is generated which can then be downloaded onto the NIOS II processor for execution.

2 Lab 0: Developing the Lab Design Platform

The Lab Instructor will inform you how to set up the initial starter files for the Labs. This will involve setting up a course directory on the workstation computer and downloading the zip folder(s) from University of Waterloo LEARN webpage for the course.

The first exercise in this lab course will be to build a processor system and then to test it with some test software. Both of these items will be designed on the workstation computer and then downloaded into the FPGA device that is resident on the LogicalStep development board. The list provided in Appendix D can be helpful in keeping track of your Lab 0 development progress.

1.2 Lab 0 Hardware Development

At this point we will start the lab by building the FPGA hardware and related peripherals.

Launch Altera Quartus Prime (Quartus 15.1) from the start menu as shown below in Figure 2.

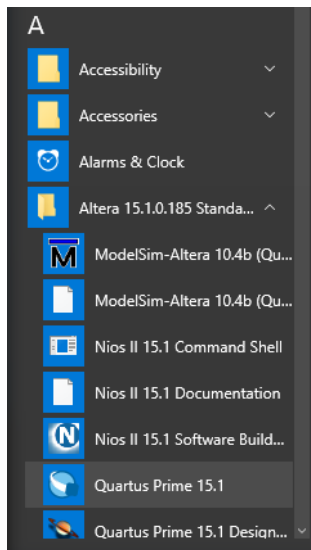


Figure 2: Launching the Altera Quartus 15.1 tools

Then within Quartus create a new project by going to the TAB: 'File>New Project Wizard...'

In the New Project Wizard set the working directory (C:\users\<userid>\ECE-224\LogicalStep) as shown in Figure 3.

**Watch out!**

Using a working directory or project name with spaces (e.g., '\My Documents\') will create serious problems that will require the project being rebuilt from scratch to fix the issues. Be very careful to ensure there are no spaces in either parameter.

Be careful to set the working directory (with your own "userid") and the top file name exactly as shown to match with the course supplied top file you will be given.

Figure 3: New Project Wizard

With the project names in place click 'Next' until you get to the 'Add Files' dialog box. At this point we need to add the course supplied project files from LEARN to the project. Use the browse button as shown in Figure 4 to add the files.

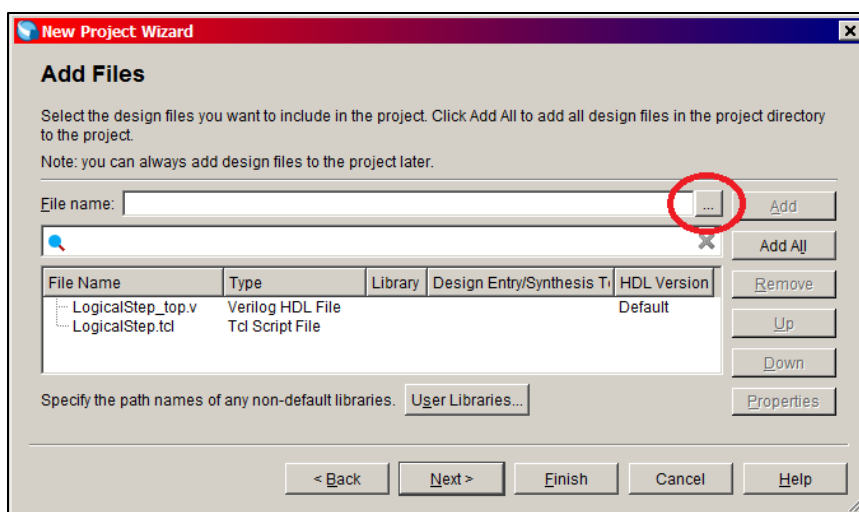


Figure 4: The 'Add Files' dialog box

Be sure to change the file filter drop down box to 'All Files (*.*)' to ensure all files are visible and add "LogicalStep_top.v" and "LogicalStep.tcl" to the project.

Click 'Next' to get to the 'Family and Device Settings' dialog box and choose the Max10 device '10M08SAE144C8G' which is the device on the LogicalStep development board. You can significantly shorten the list of devices by setting the dropdown boxes as shown in Figure 5. With the correct device selected in the 'Available devices' window click the 'Finish' button.

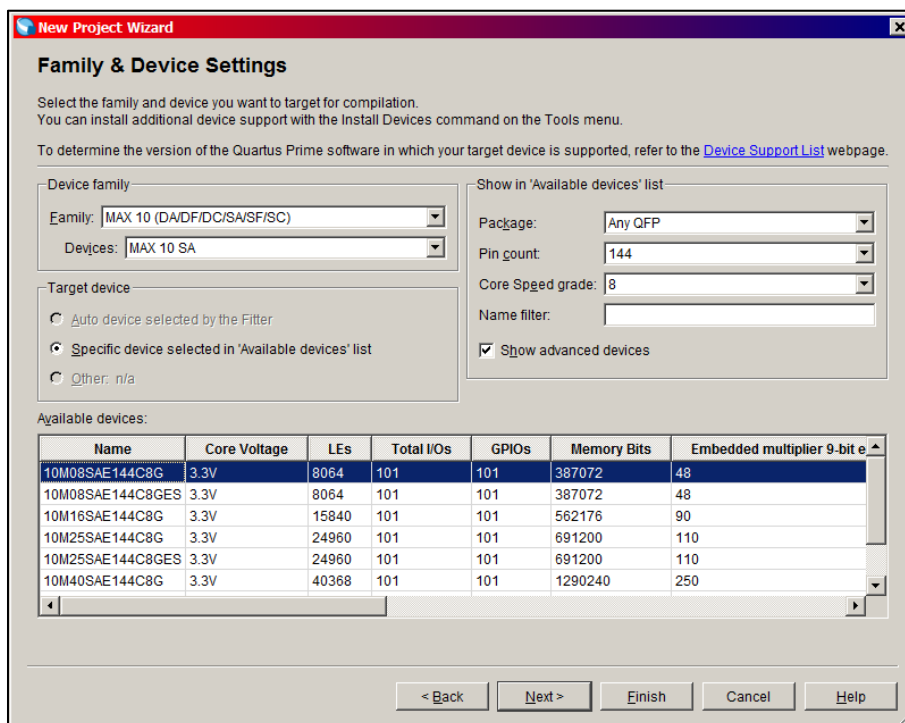


Figure 5: Family and Device Settings dialog box

Your project will now be created by Quartus. Since you provided the new project wizard with a top file name and added a matching top file to the project Quartus will automatically set the top file in the project for you.

2.1.1 The Top File

At this point let's examine the top file you were provided, double click on 'LogicalStep_top' in the 'Project Navigator' as shown in Figure 6.

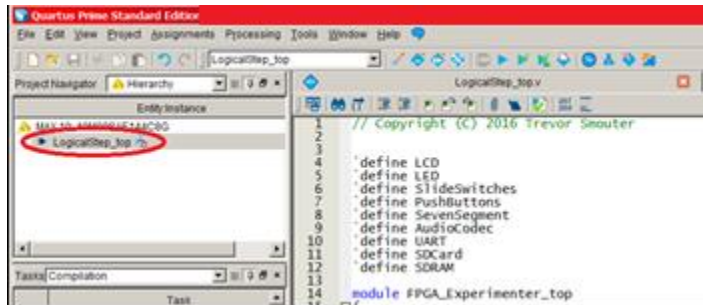


Figure 6: Selecting the top file in the Project Navigator

The top file will now be displayed (if this doesn't work then you may have made an error in naming the top file or including the top file in the project during the creation of the project).

As was mentioned previously the top file is called the top file because it is the top of the hierarchical design. It can be thought of as the main () if it was a software language. Any and all logic in your design must be instantiated in the top file to be included in the compile. In our case the top file will instantiate the generated output from Qsys (QD1) which in turn will contain a lot of different IP cores that will be included into the QD1 design later. The hierarchy for our design can be seen in Figure 7.

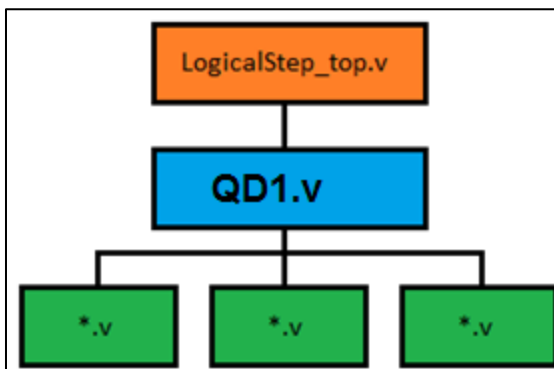


Figure 7: Hierarchy for the LogicalStep design

Upon closer inspection of the top file in Figure 6 above we see that it starts with a number of define statements. This “defines” area allows you to leave out portions of the design if you are not using some of the hardware. Also note that they indicate the different hardware sections available on the LogicalStep board that connect to the FPGA.

Below the “defines” area is the Verilog module declaration of ‘LogicalStep_top’. It is important that this name match the top file name. Inside here all of the top level ports are declared and you will notice that the only ports not within a define statement is the clock and reset at the top, every HDL design needs a clock and a reset. The rest of the ports are grouped together within their related define statement to allow easy removal of a feature should you desire to remove it. It is interesting to note that if you comment out a portion of the “defines” area such as the SDRAM, all of the pins related to the SDRAM will be removed from the design but it doesn’t stop there. During compilation Quartus will notice the missing pins and will subsequently remove everything in the design related to the SDRAM all the way back to the NIOS processor if it is included in the design. This is because Quartus can see that nothing can influence the SDRAM if the pins are missing and it is useless to have any associated hardware in the design therefore it will optimize it away.

At the bottom of the file is a space where you will instantiate your Qsys project once it is complete. We will discuss this in more detail later.

2.1.2 ***Assigning Pins***

The next step to build the project is to assign all of the IO ports to pins. Normally with a fresh project you would have to grab the board schematic and manually enter the pin numbers for each of the ports in the top file using the pin planner in Quartus. Instead we have provided a .tcl script (pronounced “tickle”) to automatically assign the port pins for you. Since you already included the file in the project you can simply go to ‘Tools>Tcl Scripts...’ select the LogicalStep.tcl file and click ‘Run’. Execution of this tcl script will automatically set the pin assignments in your project.

2.1.3 ***Qsys***

It’s now time to assemble the components of the NIOS II system and this is done in the Qsys utility. Before you open Qsys and start building the system you need to ensure that the IP folder

(that was supplied in the lab materials from the course webpage on LEARN) is in your LogicalStep project folder.



Watch out! _____

If you don't have the IP folder in your project folder then the custom IP cores that were developed for this lab will not be available in the Qsys 'IP Catalog' to add into your project. Ask the lab staff for help, if you have this situation.

To open the Qsys tool in Quartus select 'Tools>Qsys'. If you're familiar with the old Quartus SOPC Builder you'll notice that Qsys is similar but has more features and is more concerned with allowing easy plug and play of IP cores other than just processor peripherals etc. The whole FPGA project can be built in Qsys out of IP cores if desired and that is what we will do for the embedded microprocessor system labs.

With Qsys open you'll start from scratch building up the project hardware and Qsys starts with an empty project except for a clock source core. There are a few things you may have to change on every core that you add to the project so we'll go over each step first with the 'Clock Source' core. You may be required to change some of the settings for each IP core from the default settings provided. In this part of the manual there is a section for each core that needs to be added. In each cores section there is description of the internal settings (the 'Parameters') that need to be changed as well as some of the nomenclature changes required for the project to work with the software. Now is a good time to save the Qsys file, use the name 'QD1' for the Qsys project (i.e.: C:\users\<userid>\ECE-224\LogicalStep\QD1.qsys).



Tips and Tricks

Get the name right as the name of the core affects how the **software** accesses the core. If the name isn't right you may run into problems during software development.

We'll go over the settings for the clock source first to get an idea of what is required for the rest of the cores. In the 'System Contents' tab of Qsys, you will notice that the starting point in the design has to do with the primary root of the "clock tree". The root is an IP Block or core described as "Clock Source". In the 'Description' column, double click 'Clock Source' to open the IP core 'Parameters' as shown in Figure 8. Ensure that the 'Clock Frequency' is 50 MHz and then close out the Parameters tab by clicking on the associated "X" in the tab.

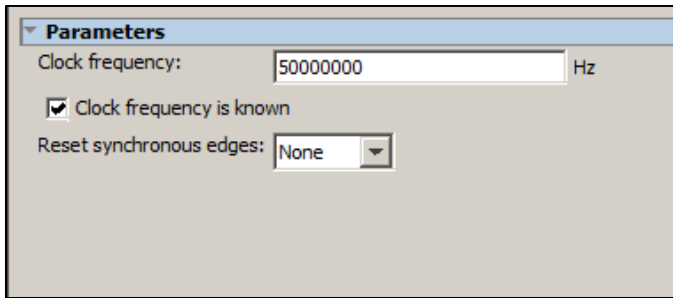


Figure 8: Clock Source Core Parameter Settings

Next you have to ensure the core's name is correct and add any required exports as shown in Table 1 for the QD1 design that we will be using. Note the table indicates: 1) the type of IP block or IP core ('Clock Source'); 2) the name that should be used to identify the IP Block in Qsys ('clk_50') for our QD1 design; 3) the exports (external pin connections to the Qsys design) if any that should be added.

Table 1: Clock Source Core Name and Exports

IP Block	Name	Export
Clock Source	clk_50	clk_in row → "clk_50" clk_in_reset → "reset"

So, from the table above change the name of the IP Block "clk_0" to 'clk_50' as shown in Figure 9 by right clicking it and selecting rename.

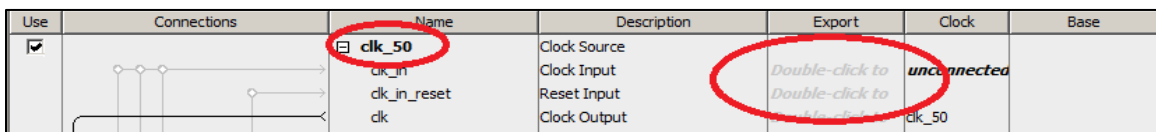


Figure 9: Naming an IP Core in Qsys

Then double click in the export column along the 'clk_in' row to add a clock input to the Qsys design. The exported pin for the clock input is to be named as shown in the table above "clk_50". The reset for the IP Block should also be exported and named as "reset".

Which should look like Figure 10 when it is finished,

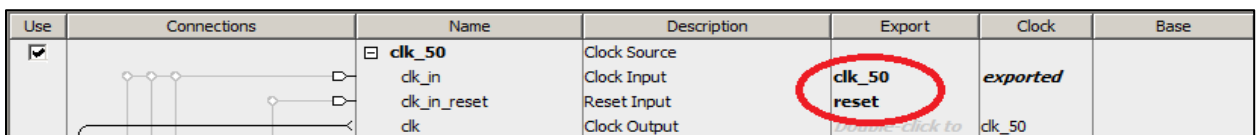


Figure 10: Export Settings of a Core in Qsys



Deep Dive

When you finish assembling your Qsys project and generate it, any exports included will become ports that will connect in the top file. These ports allow you to connect signals into and out of your Qsys hardware in the Top file. In the case of our project the top file is mainly used to connect pins on the FPGA package to signals exported from the Qsys project.

The next section directs you on how to configure each IP core block (around 20 of them) as you add it to the Qsys project. Once you configure each core block click 'Finish' on the core dialog window (NOT the Qsys window) and the IP core will be added to the project. Ensure the name and the exports are set correctly as provided in the IP Block table for each IP core shown below.



Tips and Tricks

There are two ways to find the cores you need to add in the Qsys 'IP Catalog'. The tree path has been listed under each core title, or you can use the search function with the core name.

Adding the 'Avalon ALTPLL' IP Core

(Library/Basic Functions/Clocks; PLLs and Resets/PLL)

Find the IP core in the IP catalog "Library" and double click it to add it to the project.

Set the parameters under '[1]Parameter Settings'>'General/Modes' and as shown in Figure 11

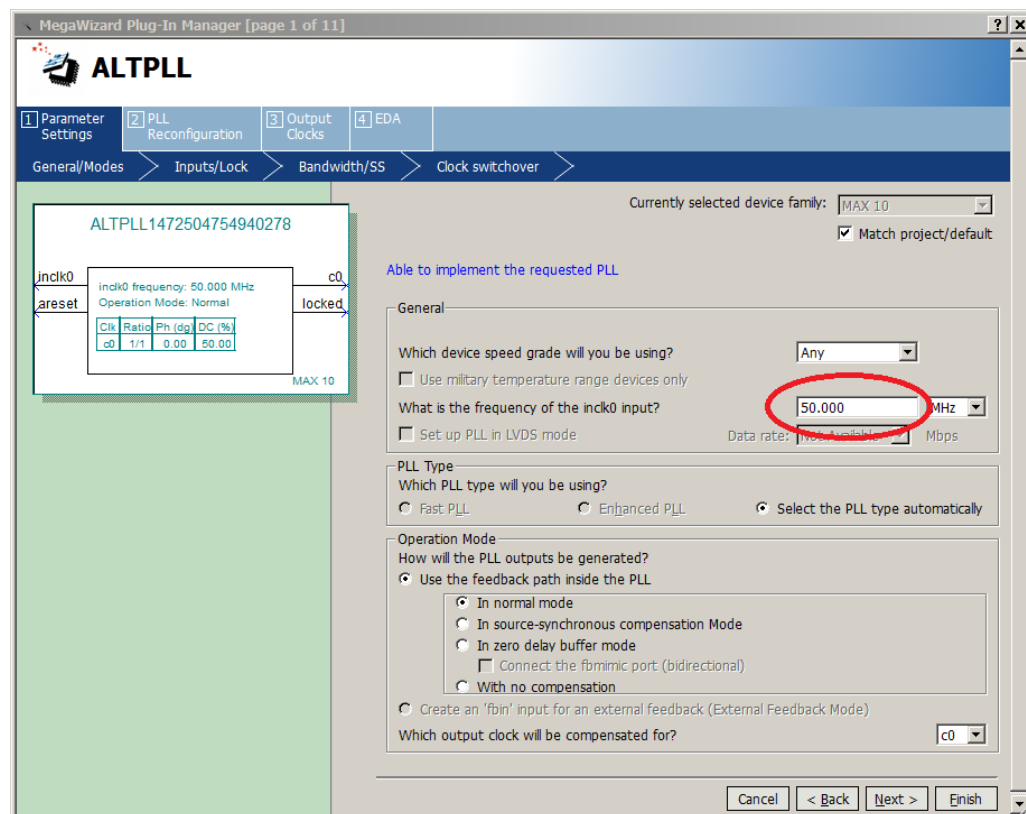


Figure 11: AltPLL Core Parameter Settings

Choose the '[3]Output Clocks' tab. The 'clk c0' sub tab comes up by default.

Set the parameters for 'clk c0' as in Figure 12. Slightly different "actual settings" values are OK.

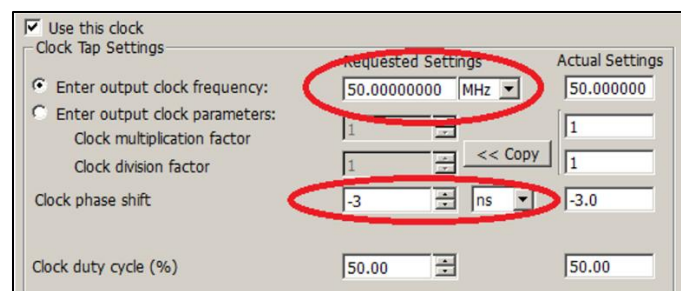


Figure 12: AltPLL C0 Parameter Settings

Choose the clk c1' tab. Set the parameters for 'clk c1' as in Figure 13.

Use this clock ☒

Clock Tap Settings

Enter output clock frequency: 12.28000000 MHz

Enter output clock parameters:

Clock multiplication factor: 1

Clock division factor: 1

Clock phase shift: 0.00 deg

Clock duty cycle (%): 50.00

Requested Settings

Actual Settings

12.280702

14

57

0.00

50.00

Figure 13: AltPll C1 Parameter Settings

Choose the clk c2' tab. Set the parameters for 'clk c2' as in Figure 14.

Use this clock ☒

Clock Tap Settings

Enter output clock frequency: 50.00000000 MHz

Enter output clock parameters:

Clock multiplication factor: 1

Clock division factor: 1

Clock phase shift: 0.00 deg

Clock duty cycle (%): 50.00

Requested Settings

Actual Settings

50.000000

1

1

0.00

50.00

Figure 14: AltPll C2 Parameter Settings

Under the '[1]Parameter Settings>Inputs/Lock' tab uncheck the 'Create locked output' as in Figure 15.

Parameter Settings

PLL Reconfiguration

Output Clocks

EDA

General/Modes

Inputs/Lock

Bandwidth/SS

Clock switchover

ALTPLL1472505676862097

indk0

areset

indk0 frequency: 50.000 MHz

Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	2/1	-54.00	50.00
c1	307/1250	0.00	50.00
c2	2/1	0.00	50.00

MAX 10

Optional Inputs

Create an 'pllena' input ☐

Create an 'areset' input ☒

Create an 'pfdena' input ☐

Lock Output

Create 'locked' output ☐

Enable self-reset on loss ☐

Figure 15: Removing the AltPll locked output



Tips and Tricks

As you add cores you'll noticed that Qsys will start reporting errors about base addresses – it is safe to ignore this as the errors will be corrected at a later step.

Then click 'Finish' (on the Core dialog window) and set the name and exports as in Table 2.

The C2 output is not exported will be used internally. Likewise, the clock input is unconnected as we will connect it to the clock_50 source later that we added previously.

Table 2: AltPll Core Name and Exports

IP Block	Name	Export
Avalon ALTPLL	<i>altpll_0 (default name)</i>	c0 → “sdram_clk” c1 → “audio_mclk”

Adding the ‘NIOS II Processor’ Module

(Library/Processor and Peripherals/Embedded Processors)

Find the IP core in the IP catalog and double-click it to open up its parameter dialog.

At this point the only setting you need to ensure is chosen is the ‘f’ variant of the processor.

Click the “Finish” button to add it to the QD1 design.

Table 3: NIOS II Core Name and Exports

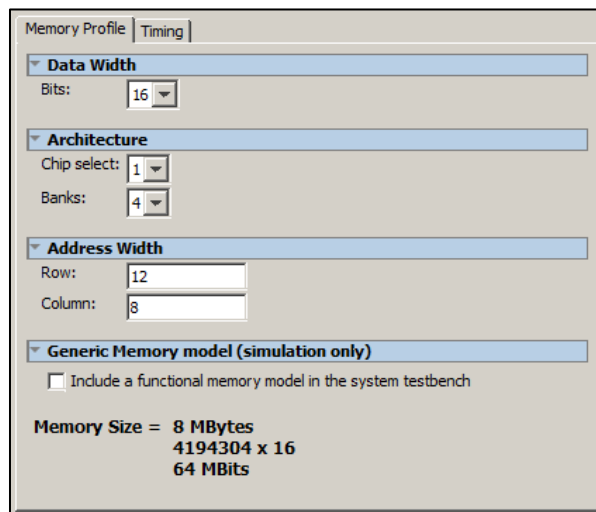
IP Block	Name	Export
NIOS II Processor	<i>nios2_gen2_0 (default)</i>	N/A

Adding the ‘SDRAM Controller’ IP Core

(Library/Memory Interfaces and Controllers/SDRAM)

Find the IP block in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 16.

**Figure 16: SDRAM Controller Core Parameter Settings**

Then click ‘Finish’ button to add it to the QD1 design and then set the name and exports as shown in Table 4.

Table 4: SDRAM Controller Core Name and Exports

IP Block	Name	Export
SDRAM Controller	sdram_0	wire → "sdram_0"

Adding the 'System ID Peripheral' IP Core

(Library/Basic Functions/Simulation/Debug and Verification/Debug and Performance)

Find the IP core and add it to the project. Just use the default settings for this IP Core.

Table 5: System ID Peripheral Core Name and Exports

IP Block	Name	Export
System ID Peripheral	<i>sysid_qsys_0 (default)</i>	N/A

Adding the 'JTAG UART' IP Core

(Library/Interface Protocols/Serial)

Find the IP core and add it to the project. Just use the default settings for this IP Core.

Table 6: JTAG UART Core Name and Exports

IP Block	Name	Export
JTAG UART	<i>Jtag_uart_0 (default)</i>	N/A

Making Qsys Connections

With these first few IP cores now in the project you can wire them up using the 'Connection' column as shown in Figure 17.

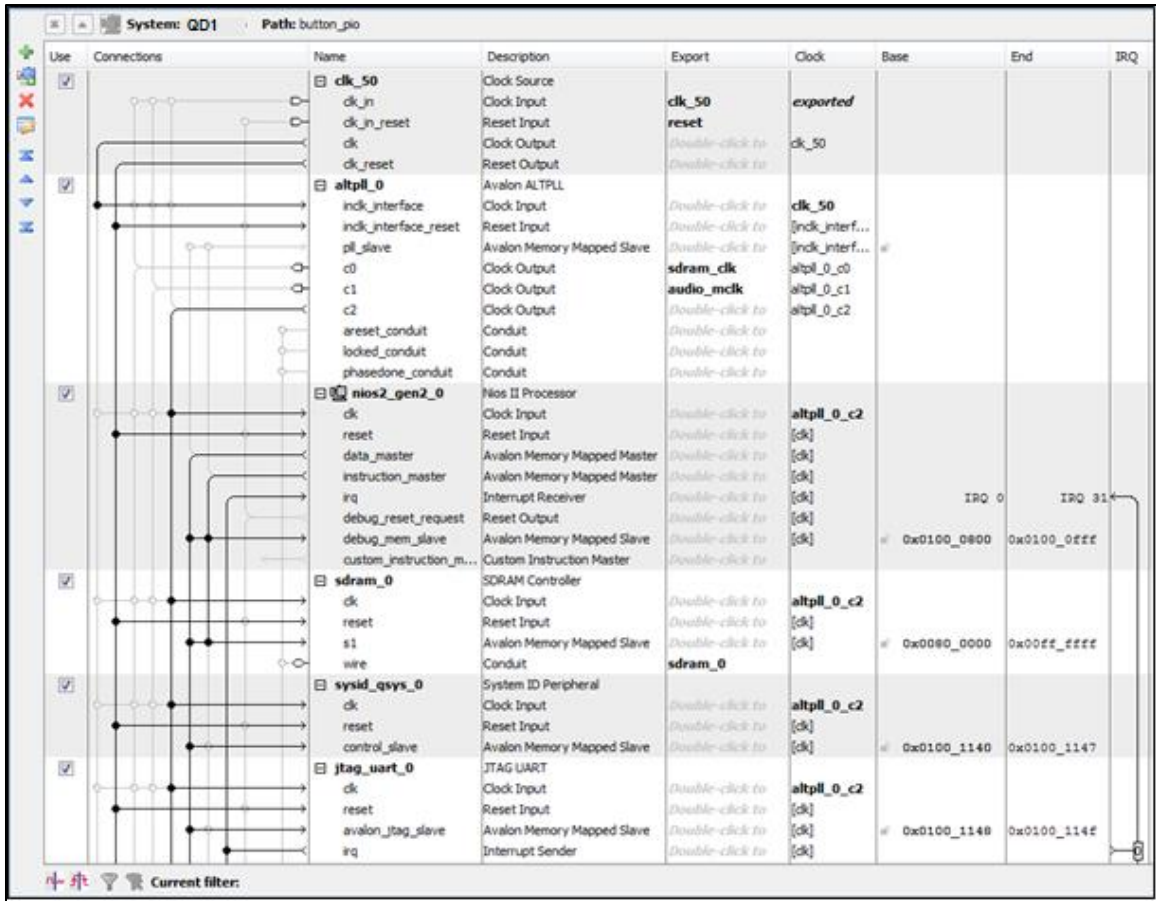


Figure 17: First Connections in Qsys

First take a moment to compare the connections column in your Qsys project to the connections column in the project shown in Figure 17. Note how the different cores have been wired together by **clicking the appropriate connection dots**. You can make the connections according to the three figures in this section that depict the connections. Alternatively you can use the 'Deep Dive' textbox below to understand why certain connections need to be made and then try do it yourself. When done, ensure your connections are correct.



Deep Dive

Clocks – Note how the 50 MHz clock comes from the top file through the 'Clock Source' IP core and passes into the 'altpll_0' which is a PLL core that provides different clocks to the processor design. The c0 clock goes to the 'sdram_clk' export and provides a phase shifted clock output to the top file which will be connected to the SDRAM memory on the board. The 'audio_mclk' export is set to support the bitrate required to provide the 44.1 kHz sampling frequency to the audio codec on the board. Finally, the c2 clock you will notice provides a 50 Mhz clock to every other core in the processor. Note that every cores clock must be connected to this.

Reset – Note how the 'clk_reset' output of the 'clk_50' Clock Source core also connects to every core in the system. This is the reset signal which is exported to the top file and will be connected to the reset button on the LogicalStep board allowing you to do a full reset of each core when you press the reset button.

There are three other signals you will be connecting to complete your system which interface with the NIOS processor.

Data_master – This is the Avalon bus connection that allows the different IP Blocks to communicate with the NIOS processor. Any block where data is sent to or from the NIOS, or is controlled by the NIOS will have a port that can connect to the NIOS' data_master.

Instruction_master – This is the NIOS connection where the NIOS accesses its program memory. Note that this is connected to the SDRAM in our project because when you download code to the board from the NIOS build tools you end up downloading it to the SDRAM. Since the NIOS also uses the SDRAM as data memory for you project you'll notice that both the instruction_master and data_master are connected to the SDRAM.

IRQ – Any cores that are set to generate interrupts must be connected to the IRQ input of the NIOS processor otherwise interrupts will not work for that core.



Tips and Tricks

Don't worry about the 'Base' and 'End' addresses shown in the Qsys window below. They will automatically be added in a later step.

Now we'll continue to add more cores to the system.

Adding the 'PIO' IP Core for LEDs

(Library/Processors and Peripherals/Peripherals)

Find the IP core and add it to the project. Use the settings as shown in Figure 18.

Figure 18: LED PIO Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 7.

Table 7: LED PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	led_pio	external_connection → "led_pio"

Adding the 'PIO' IP Core for Push Buttons

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 19.

Figure 19: Push Button PIO Core Parameter Settings

Then click 'Finish' and set the name and exports as shown in Table 8.

Table 8: Push Button PIO Core Name and Exports

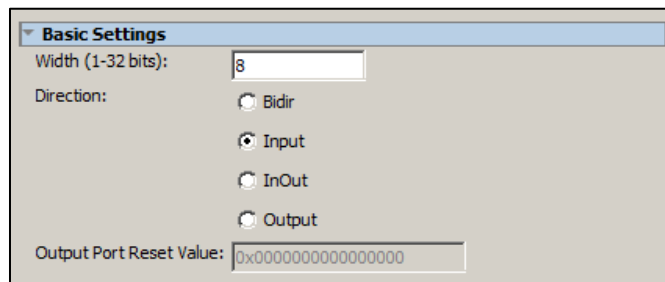
IP Block	Name	Export
PIO (Parallel I/O)	button_pio	external_connection → "button_pio"

Adding the 'PIO' IP Core for Switches

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 20.

**Figure 20: Switch PIO Core Parameter Settings**

Then click 'Finish' and set the name and exports as shown in Table 9.

Table 9: Switch PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	switch_pio	external_connection → "switch_pio"

Adding the 'Altera Avalon LCD 16207' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Just use the default parameter settings for this core but update name and exports as shown in Table 10.

Table 10: Altera Avalon LCD 16207 Core Name and Exports

IP Block	Name	Export
Altera Avalon LCD 16207	lcd_display	external = "lcd_display"

Adding the 'Audio and Video Config' IP Core

(Library/University Program/Audio & Video)

Find the IP core in the IP catalog and double click it to add it to the project.

Set the parameters as shown in Figure 21 and the name and exports as shown in Table 11.

Figure 21: Audio and Video Config Core Parameter Settings

Table 11: Audio and Video Config Core Name and Exports

IP Block	Name	Export
Audio and Video Config	audio_i2c_config	external_interface = "audio_i2c"

Adding the 'Audio' IP Core

(Library/University Program/Audio & Video)

Find the IP core in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 12. Use the default parameters for all other settings.

Table 12: Audio Core Name and Exports

IP Block	Name	Export
Audio	<u>Audio (ensure this name is capitalized to avoid problems later)</u>	external_interface → "audio_out"

Adding the 'UART (RS-232 Serial Port)' IP Core

(Library/Interface Protocols/Serial)

Find the IP block in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 13. Use the default parameters for all other settings.

Table 13: UART Core Name and Exports

IP Block	Name	Export
UART (RS-232 Serial Port)	uart	external_connection → "uart"

At this point you can make the connections for the newly inserted cores as shown in Figure 22.

NOTE: The IRQ connections must be made from the "earlier" IP Cores FIRST to assign the proper interrupt priority.

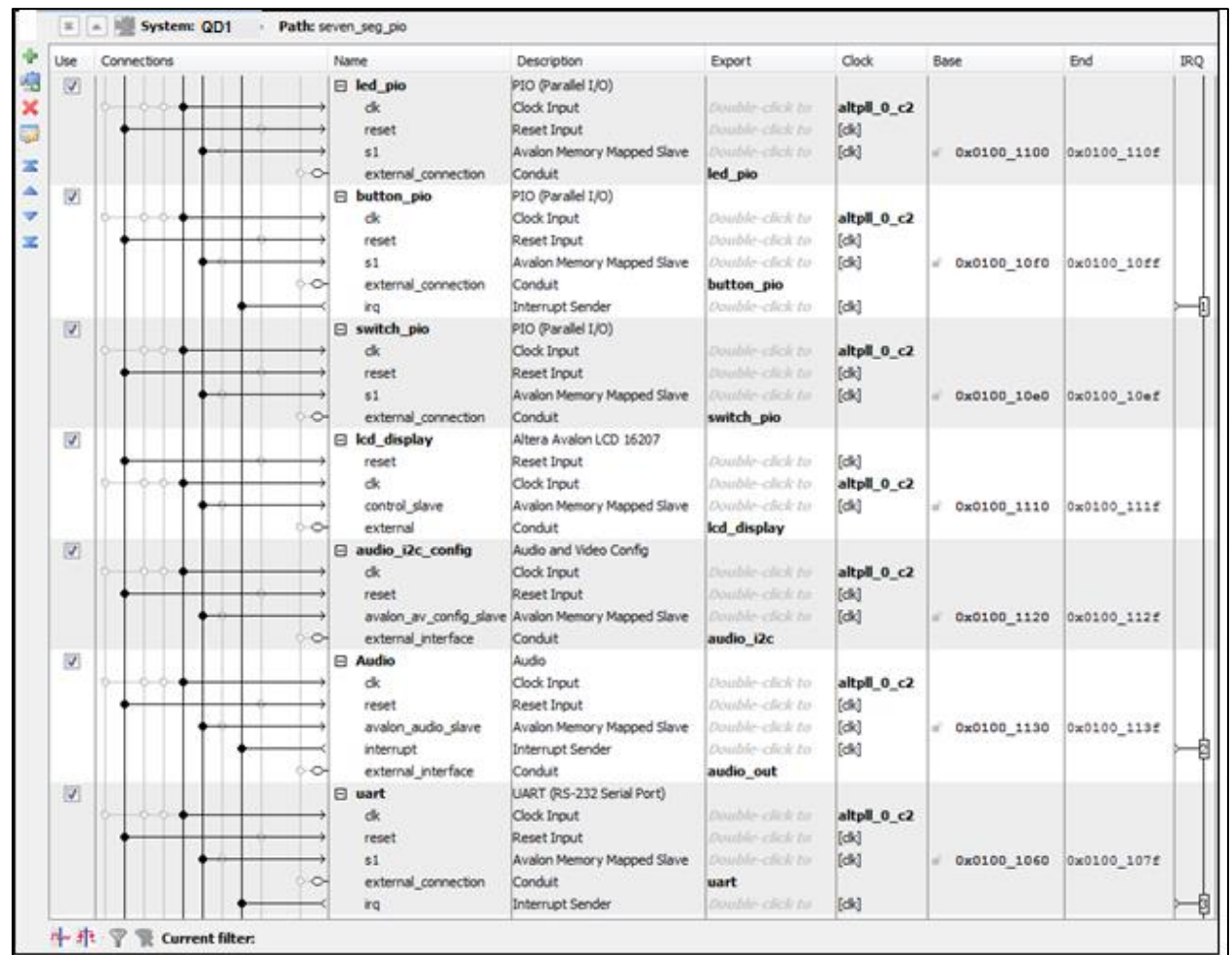


Figure 22: More Qsys Core Connections

Adding the 'Interval Timer' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Set the name and exports as shown in Table 14.

Table 14: System Timer Core Name and Exports

IP Block	Name	Export
Interval Timer	system_timer	N/A

Adding another 'Interval Timer' IP Core

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and double click it to add it to the project. Just use the default settings for this core.

Adding the 'SPI Master (3 Wire Serial)' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no parameter settings required for this core but set the name and exports as shown in Table 15.

Table 15: SPI Master Core Name and Exports

IP Block	Name	Export
SPI Master (3 wire serial)	spi_master	external → "spi_master"

Adding the 'Dual 7 Segment' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no parameter settings required for this core but set the name and exports as shown in Table 16.

Table 16: Dual 7 Segment Core Name and Exports

IP Block	Name	Export
Dual 7 Segment	seven_seg_pio	dual_7_segment → "segment_drive"

Adding the 'EGM' IP Core

(Project)

Find the IP core in the IP catalog and double click it to add it to the project. There are no settings required for this module but set the name and exports as shown in Table 17.

Table 17: EGM Core Name and Exports

IP Block	Name	Export
EGM	egm	interface → "egm_interface"

Adding the 'PIO' IP Core for Stimulus_in

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and add it to the project. Set the parameters as shown in

Figure 23 (**NOTE the EDGE TYPE set to RISING**) and then name the exports as shown in Table 18.

Basic Settings

Width (1-32 bits): 1

Direction: ☐ Bidir ☒ Input ☐ InOut ☐ Output

Output Port Reset Value: 0x0000000000000000

Output Register

☐ Enable individual bit setting/clearing

Edge capture register

☒ Synchronously capture

Edge Type: RISING

☐ Enable bit-clearing for edge capture register

Interrupt

☒ Generate IRQ

IRQ Type: EDGE

Level: Interrupt CPU when any unmasked I/O pin is logic true
Edge: Interrupt CPU when any unmasked bit in the edge-capture register is logic true. Available when synchronous capture is enabled

Figure 23: Stimulus in PIO Core Parameter Settings

Table 18: Stimulus in PIO Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	stimulus_in	external_connection → "stimulus_in"

Adding the 'PIO' IP Core for Response_out

(Library/Processors and Peripherals/Peripherals)

Find the IP core in the IP catalog and add it to the project. Set the parameters as shown in Figure 24 and the name and exports as shown in Table 19.

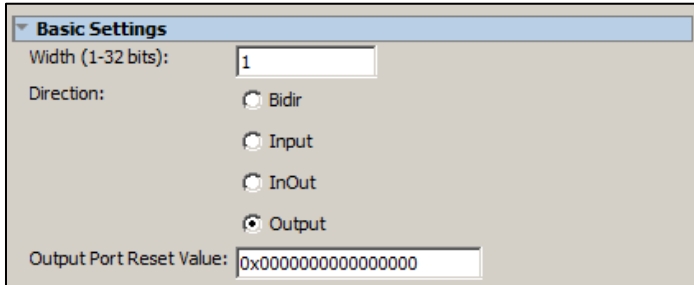


Figure 24: Response out PIO Core Parameter Settings

Table 19: Response out Core Name and Exports

IP Block	Name	Export
PIO (Parallel I/O)	response_out	external_connection → "response_out"

At this point you can make the connections for the rest of the core as shown in Figure 25.

NOTE: The IRQ connections must be made from the "earlier" IP Cores FIRST to assign the proper interrupt priority.

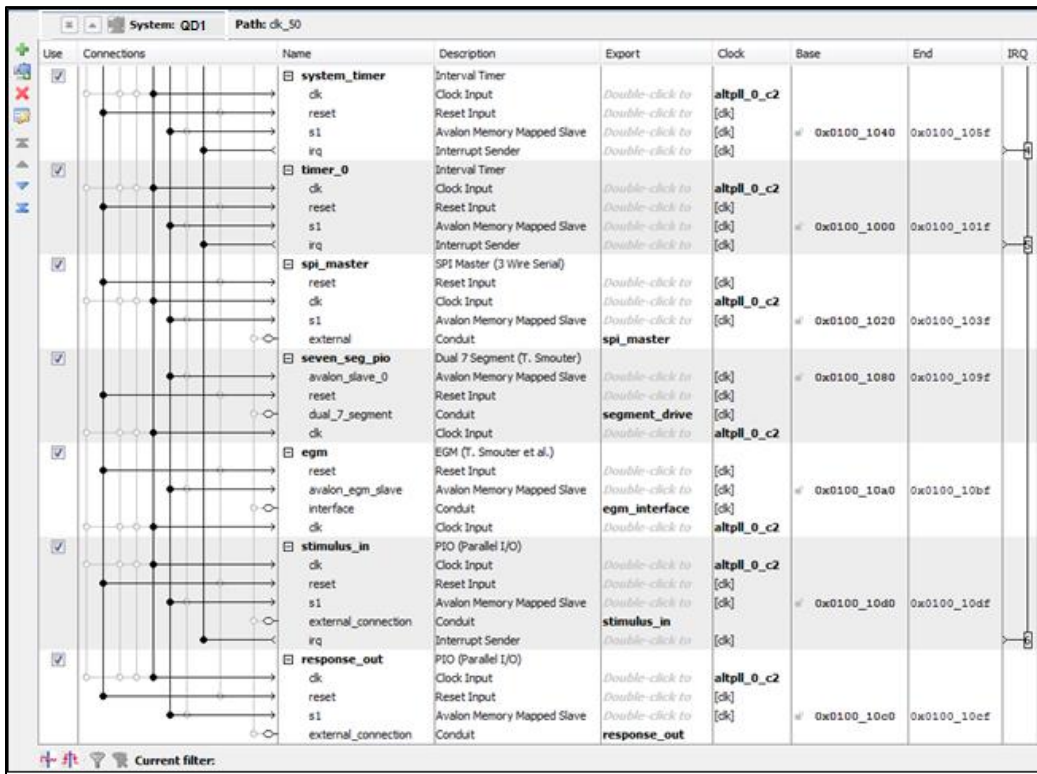


Figure 25: Remaining Qsys Connections in Qsys Window

Now you should set the base addresses of each core so that each core has a unique address that doesn't overlap with other cores. Qsys has the ability to set all the base addresses automatically without errors which saves considerable time for the designer. Just select the Tab option in Qsys: 'System>Assign Base Addresses' to assign the base addresses.

With all the cores now added and the Base Addresses now set for the IP Cores in the Qsys project you can set the memory vectors in the NIOS processor by double clicking on the processor core (added in section 0), select the 'Vectors' tab and adjust the settings as shown in Figure 26.

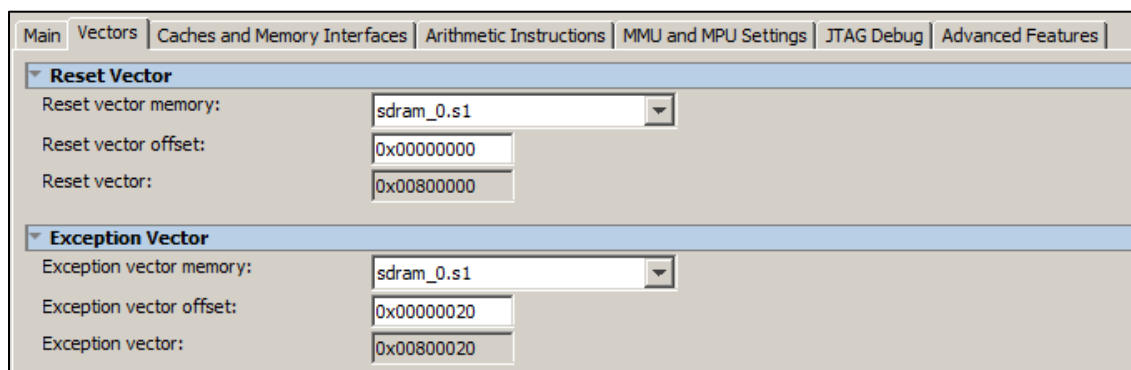


Figure 26: NIOS Memory Vector Parameters

Hopefully everything is now ready to go. Click the 'Generate HDL...' button in the bottom right hand corner of the Qsys window. After the 'Save' dialog is finished and closed the system will show the Generation Window as seen in Figure 27. Please ensure that "QD1" has been included at the end of the Output Directory field.

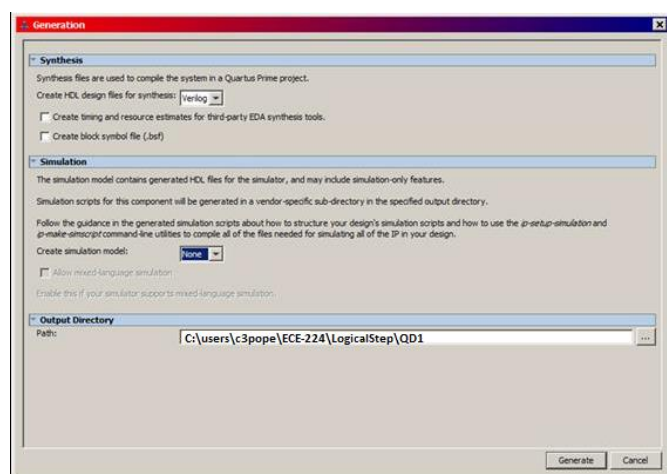


Figure 27: Qsys Project Generation

Press 'Generate' as seen in Figure 27 and when it completes press the "Close" button. Then terminate the QSYS processing by clicking on "FINISH" button.

2.1.4 *Editing the Top File to Instantiate the Qsys Project*

In the 'Project Navigator' right click on the LogicalStep_top.v file and select 'Set as Top Level Entity' which sets the provided top file to be the top file according to Quartus. Click the dropdown in the 'Project Navigator' and set it to 'Hierarchy' then double click 'LogicalStep_top' to open the top file.

Scroll down to the bottom of the top file to the place where it says "Place Qsys instance below here". Now you need to get the automatically generated Qsys instance template which can be found at 'File>Open>QD1>QD1_inst.v'. Copy the Verilog instantiation text that is in this file and paste it into the top file between the comments.

Now you need to assign the top level ports (which are in fact the pins on the FPGA that connect to the different interfaces on the LogicStep board) to the Qsys module you created. This is done by replacing the corner bracket text in the instantiation code to the proper input or output ports in the top file. For instance the following line:

```
.clk_50_clk      (<connected-to-clk_50_clk>),
```

Needs to be changed to:

```
.clk_50_clk      (clkin_50),
```

Where clkin_50 is an input signal that can be found close to the top of the top file. Go ahead and assign all the inputs and outputs in this manner trying your best to match up the signals provided in the top file.

The connections for the SPI (serial peripheral interface) are not intuitive, therefore the assignments are provided to you here:

```
.spi_master_cs      (sd_dat3),      // spi_master.cs
.spi_master_sclk    (sd_clk),       //      .sclk
.spi_master_mosi     (sd_cmd),      //      .mosi
.spi_master_miso     (sd_dat0),     //      .miso
.spi_master_cd       (),           //      .cd
.spi_master_wp       (),           //      .wp
```

There are two 'wires' in the top file called "stimulus" and "response". These wires are for connecting .egm_interface_stimulus to .stimulus_in_export and .egm_interface_response to .response_out_export as these signals come to the top file only to connect these cores together – these signals do not leave the FPGA.

2.1.5 *Compiling the Project*

At this point you should be able to compile the project. Select 'Processing>Start Compilation' to compile the project and cross your fingers for a successful compile. If you run into trouble here please ask the lab instructor or TAs for help.

2.1.6 *Hardware Wrap-up*

With the hardware design complete the next step is to configure the FPGA on the board with the image that was generated by Quartus during the compile procedure. The Quartus programmer can be accessed from the menu bar at 'Tools>Programmer'. With the programmer window open ensure the LogicalStep board is detected by the programmer as shown in Figure 28.

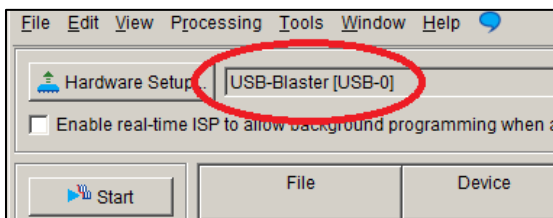


Figure 28: Quartus Programmer Setup with USB cable connected

If you do not see a USB-Blaster connected click the 'Hardware Setup...' button and double click the USB-Blaster from the list of 'Available Hardware Items' and close the 'Hardware Setup' window. If this does not work then ask the Lab Instructor for assistance.

Click the 'Add File' button on the left side of the Programmer window, navigate to the 'output_files' folder within your project folder and select the **.sof** file generated by Quartus during the compile. Ensure the 'Program/Configure' checkbox is checked and then click the 'Start' button to configure the FPGA with the hardware design you generated in Quartus. Once the FPGA is successfully configured, you have actually turned the logic building blocks within the FPGA into a hardware arrangement that encompasses all of the logic required for the FPGA to be a complete microprocessor – known as a soft core processor (NIO II) system. The next step is develop the software code that will run in the NIO II processor.

2.2 Lab 0 Test Software Development

At this point it is time to start working with the NIOS II Software Build Tools (SBT), using the Altera Eclipse IDE (Integrated Development Environment), discussed earlier to generate software code to exercise the hardware via the NIOS II processor. Refer to Figure 29 below.

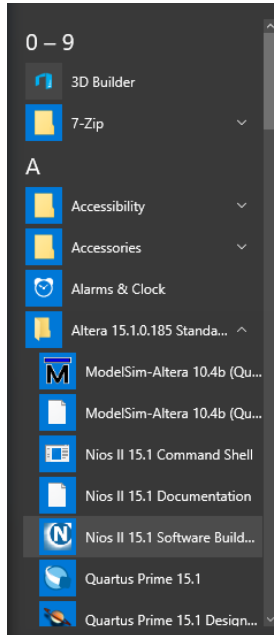


Figure 29: Launching the Altera NIOS II Software Build Tools

The Altera Eclipse environment has a number of built-in templates for testing NIOS II systems. One template is called Board Diagnostics. That one will be used to initially test the hardware you built earlier in Quartus/Qsys. It is designed to allow users to exercise the NIOS II processor and some of the peripherals from a console interface within the NIOS II Software Build Tools. Initially the Board Diagnostics program provides a quick and easy way to learn how to build a software project to confirm that the hardware you assembled earlier in Quartus /Qsys is functional.

2.3.1 *Board Diagnostics Project in the NIOS II Software Build Tools*

Start by launching the NIOS II Software Build Tools from the Windows start menu – it is best to navigate through ‘All Apps’ to the Altera folder instead of using the Start menu search function as it is easy to find the wrong program using this approach. When you open Eclipse for the first time it will ask you for your workspace.

Browse to your project folder (C:\users\<userid>\ECE-224\LogicalStep) and create a subfolder named 'software' within your project folder and set this as your workspace (like that shown in Figure 30).

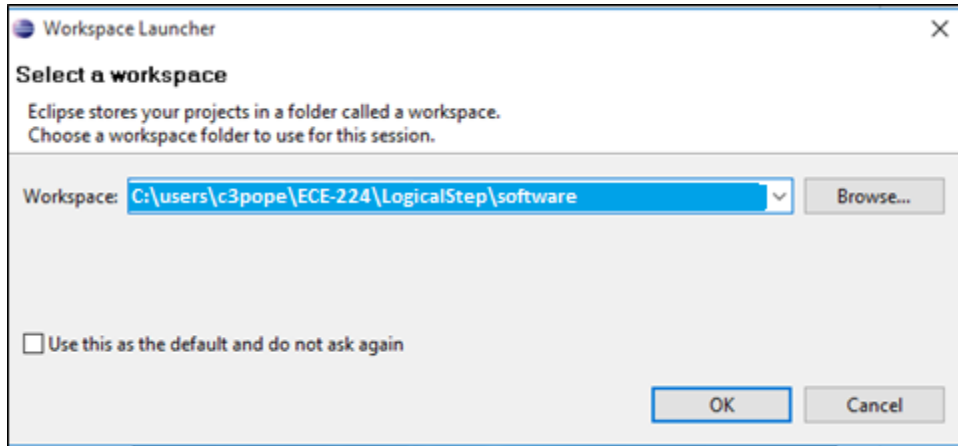


Figure 30: Eclipse Workspace Setup



Watch out! —————

It is important that you don't change your workspace relative to your project folder or use a copy of your project folder moved to another location or rename your project folder. Eclipse remembers file paths to make things easier but if you move things around it's easy to break the project which can be difficult to fix because all the default paths are wrong.

Press OK and then the NIOS II – Eclipse window will appear as in Figure 31.

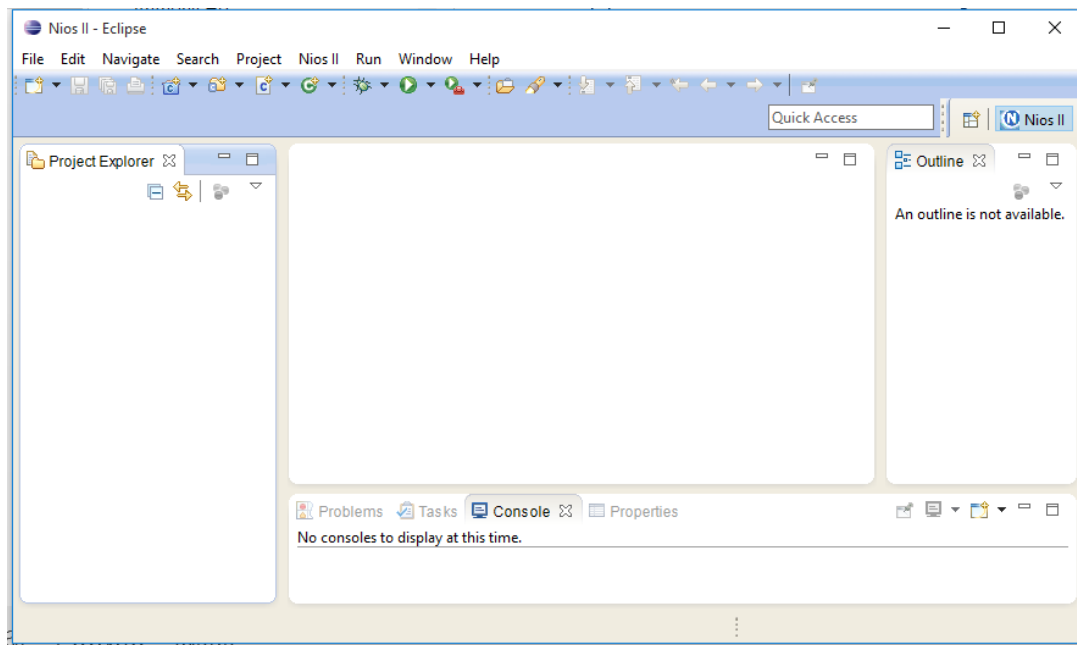


Figure 31: NIOS II- Eclipse Window

To create the board diagnostics project in Eclipse go to the tab “File” and then select ‘File>New>Nios II Application and BSP from Template’. The example dialog window as shown in Figure 32 will appear.

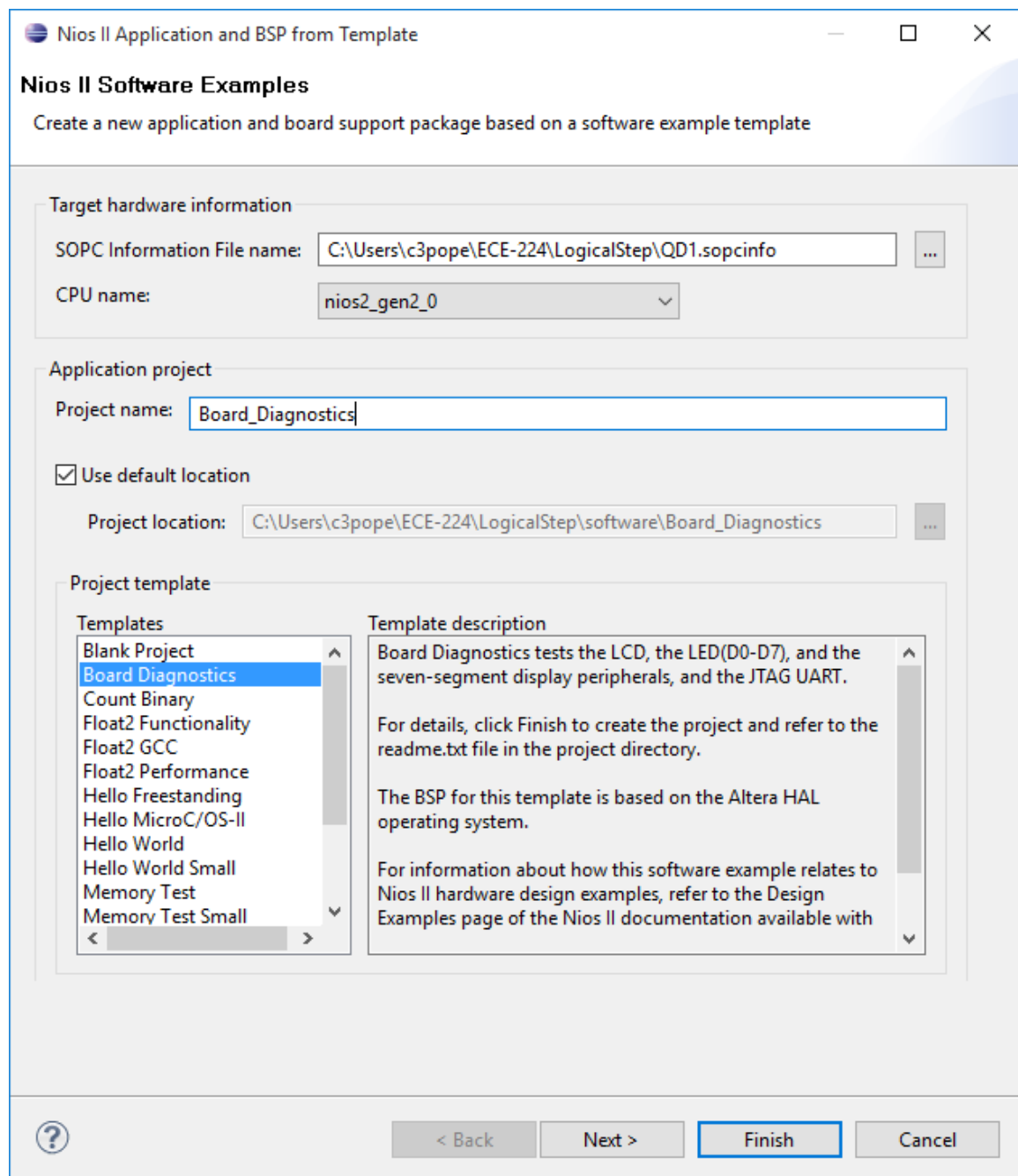


Figure 32: NIOS II Application and BSP from Template option

In the dialog window that opens you need to BROWSE to the project's Qsys-generated .sopcinfo file for insertion into the SOPC Information File Name field. If you have followed the earlier directions properly then the file to select here at this step is the QD1.sopcinfo file located in your project directory. After the Eclipse is done loading the QD1 .sopcinfo file (which can take a few seconds) the CPU name field will also be populated with the name of your NIOS II processor as defined in Qsys.

Recall that the .sopcinfo file contains all the addresses and details about the hardware that you defined in Qsys and is required by Eclipse so that your software can interact with the hardware appropriately – more on this will be discussed later.

Give your software project a name in the 'Project name' field such as "Board_Diagnostics", select the 'Board_Diagnostics' template in the 'Project Template' section and click Finish to create the project. Once the project creation is completed you'll see two projects in the 'Project Explorer' pane on the left. One is the project itself and the other is the BSP for the project. See Figure 33.

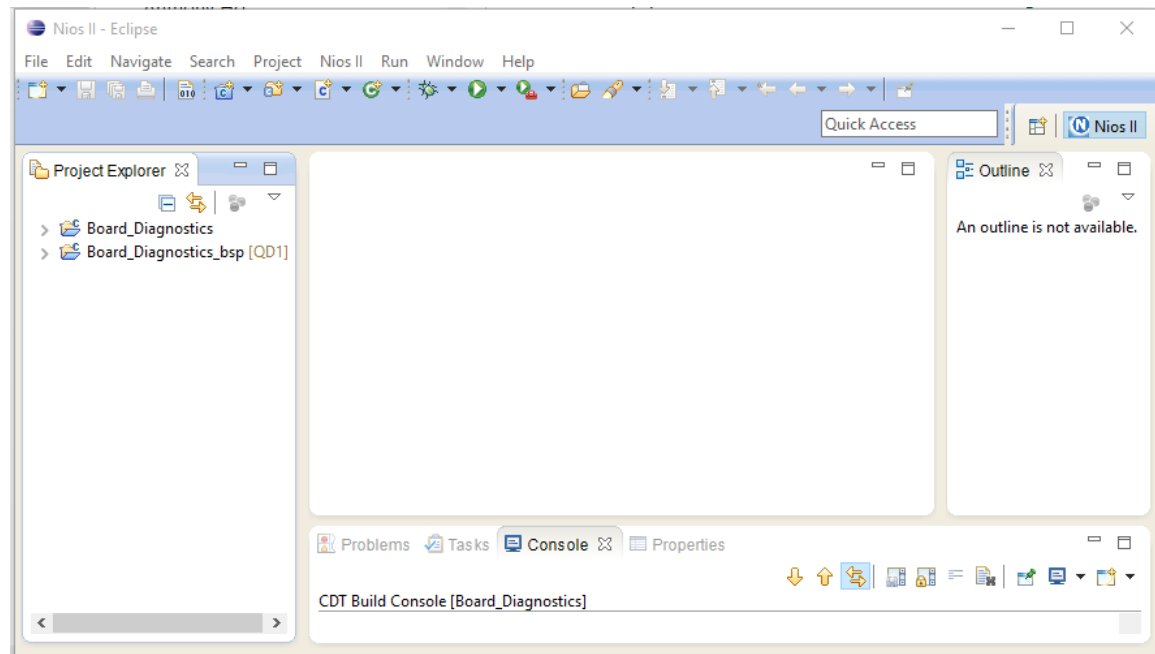


Figure 33: NIOS II Project Creation

Later on if at any time you create a new software project or if you change the Qsys hardware and recompile in Quartus you will need to regenerate the BSP. This is done by selecting the BSP project in the 'Project Explorer' pane (shown above) and then right-clicking and selecting 'Nios II>Generate BSP'.

If you haven't already downloaded the FPGA configuration file (*.sof) using Quartus Programmer (in the previous Hardware Wrap-up section) then please complete that step before proceeding below.

To compile the project, BSP project and also download the *.elf (software program) to the NIOS II processor that resides on the previously configured FPGA simply right click on the project

(Board_Diagnostics in this case) in the 'Project Explorer' pane and then select 'Run As>NIOS II Hardware'. Once the *.elf downloads to the NIOS II the 'Console' window at the bottom of Eclipse will display a menu that allows you to test the LEDs, LCD, pushbuttons, and the seven segment display. If you are missing one of these menu items it means there is a problem with the hardware design— either the hardware block wasn't added in Qsys, the QD1_inst.v was incorrectly connected or the name is incorrect in Qsys. Take a few minutes now to test each of the hardware components and ensure they are working properly.

The end goal of Lab0 is to confirm that your FPGA design is confirmed to be functional and that you can create a software program and download it to the FPGA NIOS II with successful execution.

2.4 Lab 0: Demo

Before the end of the Lab 0 session you will be asked to download and demonstrate the "Board Diagnostics" application operation for the FPGA and Nios II system. The marking scheme will be as follows:

- 1) 1 Mark: for successful downloading of the LogicalStep_top.sof to the FPGA
- 2) 1 Mark: for successful Board_Diagnostics download to the LogicalStep NIOS II memory
- 3) 1 mark: for successful Board_Diagnostics operation of the LEDs and SevenSegment Displays
- 4) 1 mark: for successful Board_Diagnostics operation of the LCD Display

2.5 Lab 0 Post-Lab Exercises: Getting to know the NIOS/Eclipse IDE

Before we send you off into the wild world of embedded development for your next lab session deliverable it is worthwhile to take a short tour of the IDE environment and some of the resources available to you so you are better equipped for success.

We'll start with a description of how the hardware that you built can be controlled from within the software code. The process of creating a project (and pointing the environment to the .sopcinfo file) is the first step and generating the BSP is the second step – as we did above.

Let's take a quick look at the result of these actions and how it can help us. In the Eclipse 'Project Explorer' pane click the '>' to the left of the 'Board_Diagnostics_bsp' to expand the "bsp" project you just built. Then expand the file 'system.h' and you will see a long list of define statements in the 'Project Explorer' pane. This is shown in Figure 34.

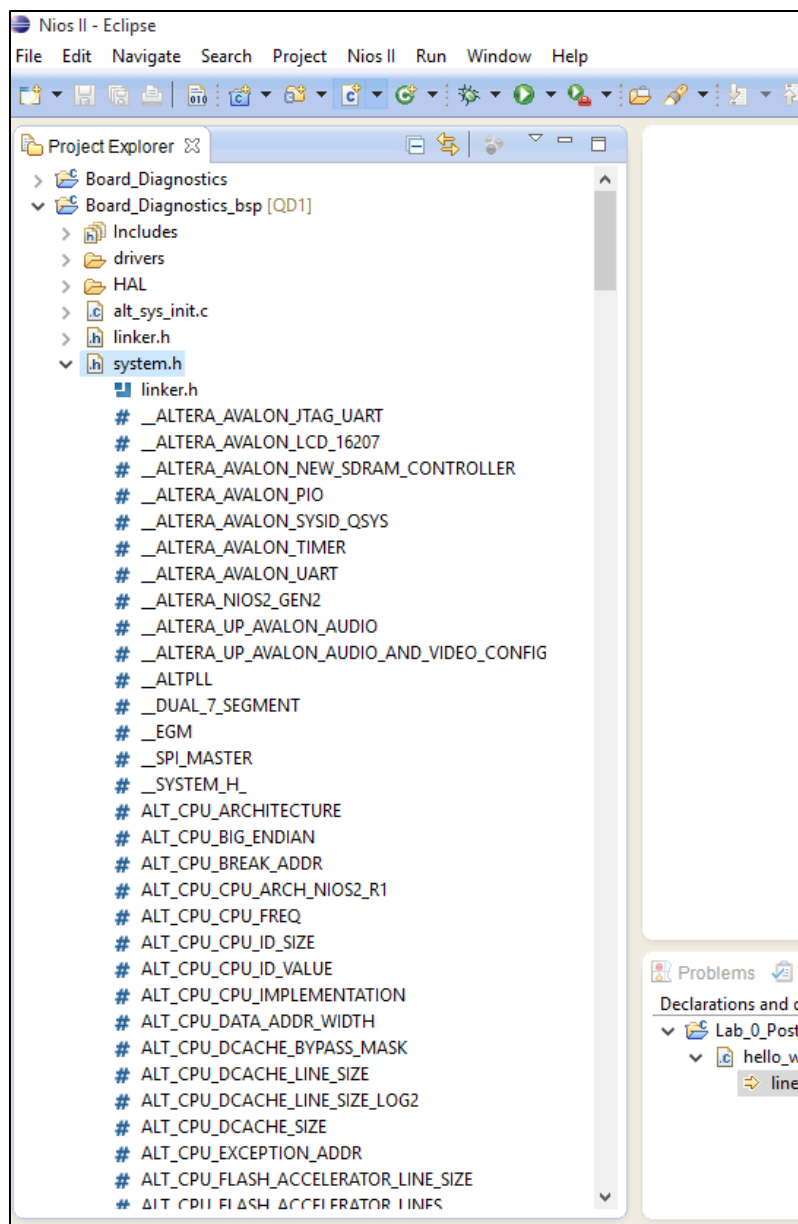
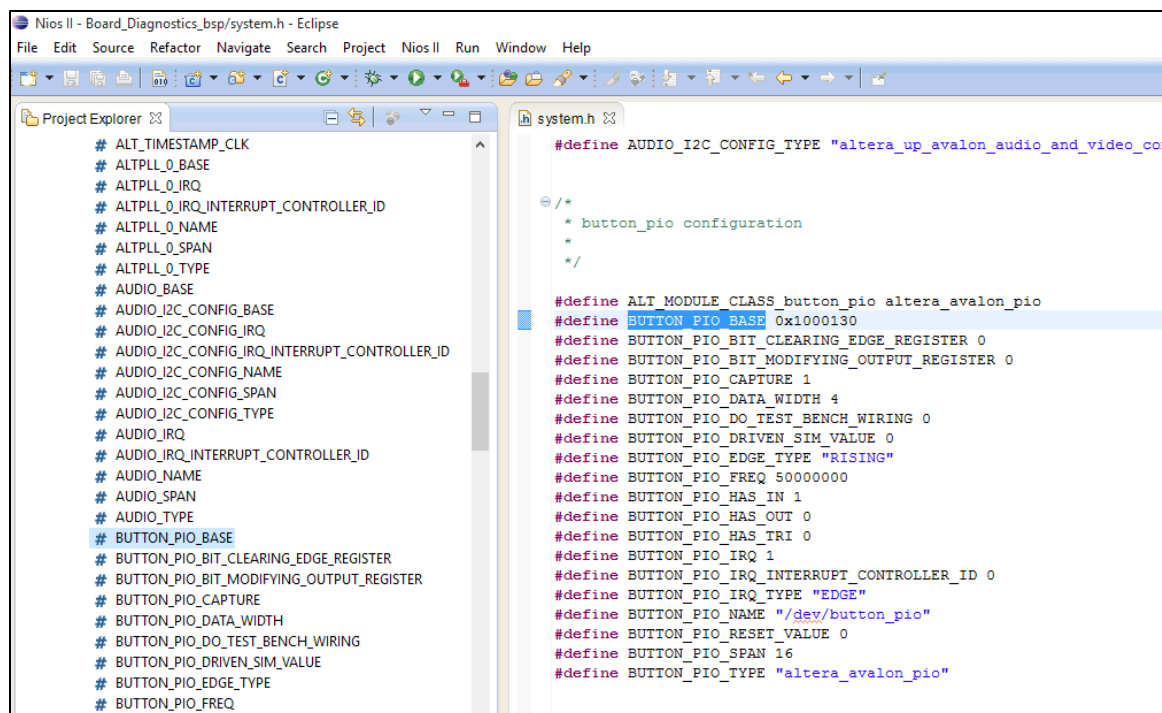


Figure 34: Referencing the system.h info from the BSP Project

Scroll down the list to find the define 'BUTTON_PIO_BASE' and double click it (see Figure 35). You'll notice that the 'BUTTON_PIO_BASE' represents a long hex number which is actually the hardware address of the button PIO. So, in your code when you want to access/refer to the push buttons your software code uses this base address value.



```

# ALT_TIMESTAMP_CLK
# ALTPLL_0_BASE
# ALTPLL_0_IRQ
# ALTPLL_0_IRQ_INTERRUPT_CONTROLLER_ID
# ALTPLL_0_NAME
# ALTPLL_0_SPAN
# ALTPLL_0_TYPE
# AUDIO_BASE
# AUDIO_I2C_CONFIG_BASE
# AUDIO_I2C_CONFIG_IRQ
# AUDIO_I2C_CONFIG_IRQ_INTERRUPT_CONTROLLER_ID
# AUDIO_I2C_CONFIG_NAME
# AUDIO_I2C_CONFIG_SPAN
# AUDIO_I2C_CONFIG_TYPE
# AUDIO_IRQ
# AUDIO_IRQ_INTERRUPT_CONTROLLER_ID
# AUDIO_NAME
# AUDIO_SPAN
# AUDIO_TYPE
# BUTTON_PIO_BASE
# BUTTON_PIO_BIT_CLEARING_EDGE_REGISTER
# BUTTON_PIO_BIT_MODIFYING_OUTPUT_REGISTER
# BUTTON_PIO_CAPTURE
# BUTTON_PIO_DATA_WIDTH
# BUTTON_PIO_DO_TEST_BENCH_WIRING
# BUTTON_PIO_DRIVEN_SIM_VALUE
# BUTTON_PIO_EDGE_TYPE
# BUTTON_PIO_FREQ

#define AUDIO_I2C_CONFIG_TYPE "altera_up_avalon_audio_and_video_co

/*
 * button_pio configuration
 */

#define ALT_MODULE_CLASS_button_pio altera_avalon_pio
#define BUTTON_PIO_BASE 0x1000130
#define BUTTON_PIO_BIT_CLEARING_EDGE_REGISTER 0
#define BUTTON_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define BUTTON_PIO_CAPTURE 1
#define BUTTON_PIO_DATA_WIDTH 4
#define BUTTON_PIO_DO_TEST_BENCH_WIRING 0
#define BUTTON_PIO_DRIVEN_SIM_VALUE 0
#define BUTTON_PIO_EDGE_TYPE "RISING"
#define BUTTON_PIO_FREQ 50000000
#define BUTTON_PIO_HAS_IN 1
#define BUTTON_PIO_HAS_OUT 0
#define BUTTON_PIO_HAS_TRI 0
#define BUTTON_PIO_IRQ 1
#define BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID 0
#define BUTTON_PIO_IRQ_TYPE "EDGE"
#define BUTTON_PIO_NAME "/dev/button_pio"
#define BUTTON_PIO_RESET_VALUE 0
#define BUTTON_PIO_SPAN 16
#define BUTTON_PIO_TYPE "altera_avalon_pio"

```

Figure 35: Examining the Button_PIO_Base Address in the system.h info



Deep Dive

If you go back to Quartus, open Qsys and find the button_pio block you'll note that in the 'Base' column for that IP the address displayed is the same as the one reported in the system.h file. The step of generating the BSP pulls all the Qsys base addresses into Eclipse.

Going forward you'll want to refer to the system.h file anytime you want to access new hardware to find the definition (or name) for its base address and other specific information.

As an informative exercise let's examine where the software c-code is located for running the "Board_Diagnostics application project. For the "canned" Board_Diagnostics application that was created earlier from a template the software that is executed by the NIOS II processor can be accessed. Go to the Eclipse 'Project Explorer' pane and click the '>' to the left of the 'Board_Diagnostics' project to expand it. Then double click on the board_diag.c file. This will open up the source c-code for the Board Diagnostics application (see Figure 36). Also note in the Project Pane that if you expand the "board_diag.c" file the individual C-code function calls/routines are listed.

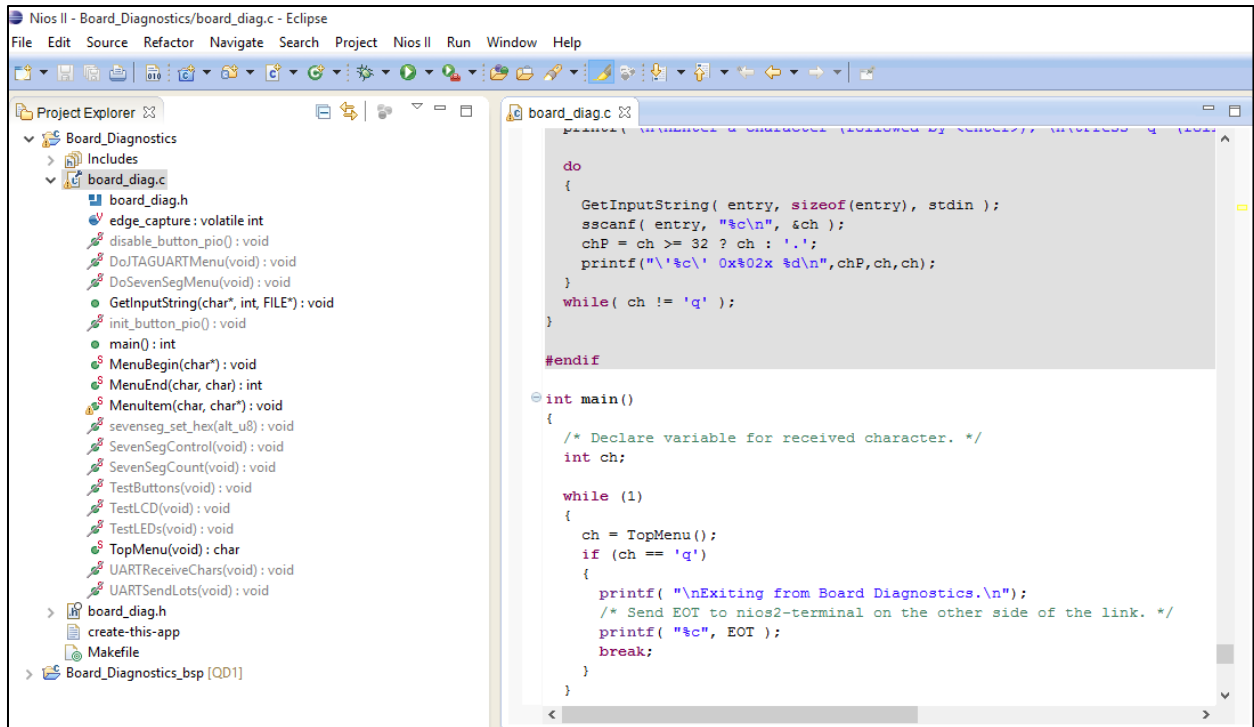


Figure 36: Looking at the Board_Diagnostics Source Code

For any of your upcoming labs the “NIO S II Application and BSP from Template” selection will be with the “Hello_World” template as a starting point for your c-code development.

2.7.1 Lab 0 Post-Lab Exercises: Setting up a Simple Read/Write Operations Project

Now that you have verified the functionality of your hardware using the Board Diagnostics in the template project in your Lab 0 Demo, you can move on and create a new simple project as a Post - Lab 0 exercise. You are encouraged to gain some experience from this exercise before the beginning of your next Lab Session.

1. Close out (do not delete) the two previous project components (Board_Diagnostics and Board_Diagnostics_BSP) within the Project Explorer pane in the IDE tool. Do this by selecting each one and then right-clicking on those items and select “Close Project”.
2. From the top menu select File > New > NIO S II Application and BSP from Template
3. Select your .sopcinfo file as you did for the “Board_Diagnostics” project (QD1.sopcinfo)
4. Give your new project a name with no spaces (“Lab_0_Post_Lab”)
5. Select the Hello World project from the Templates list as shown in Figure 37.

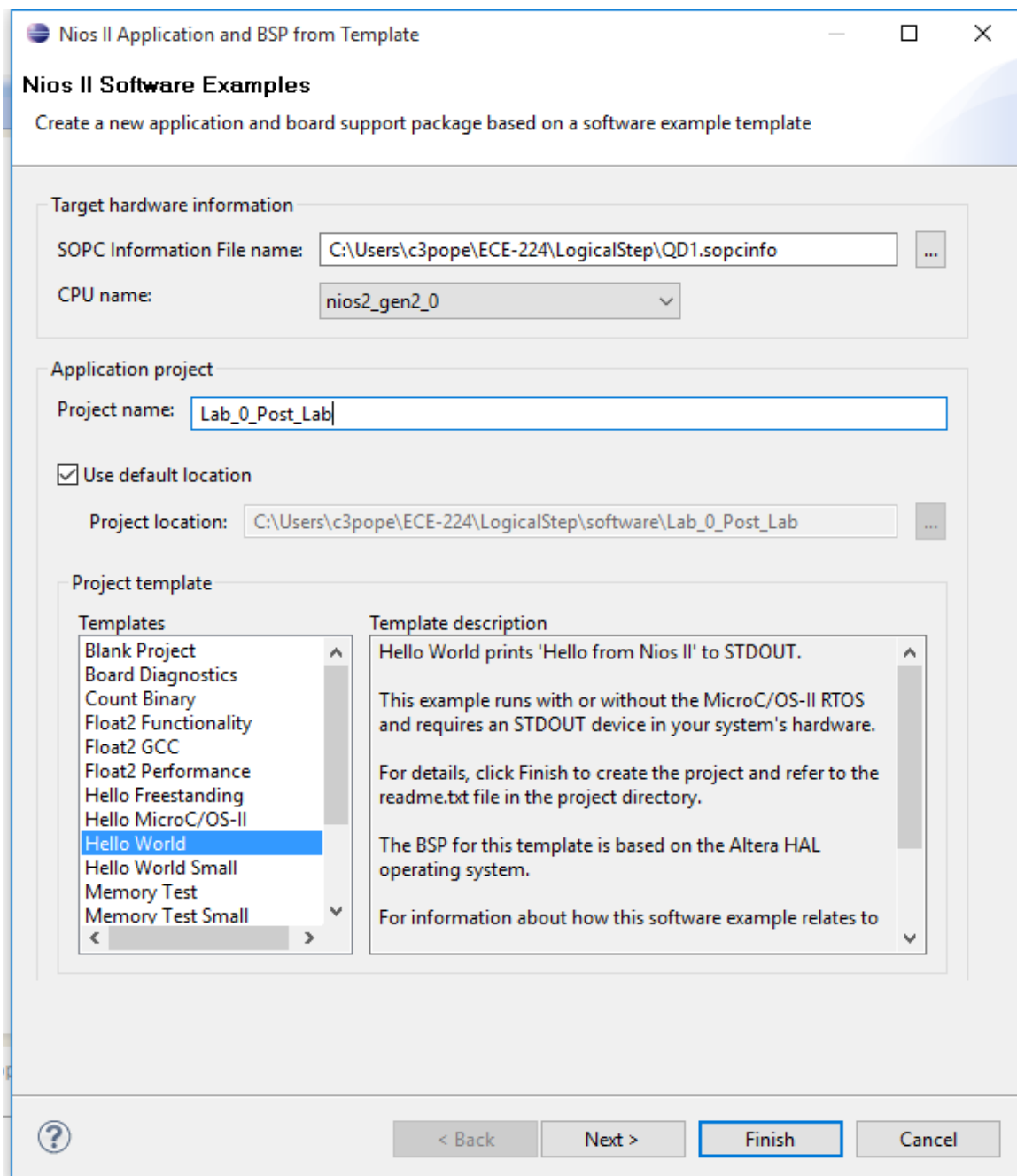


Figure 37: Configuring a New Application from a Template for Lab_0_Post_Lab

6. Click **Next**
7. Refer to Figure 38 to create the new BSP project to be coupled with the Lab_0_Post_Lab project. Click on the “Select a new BSP project based on the application project template”. Let the project BSP file default to the Lab_0_Post_Lab_bsp value. Leave the “Use default location” box selected.

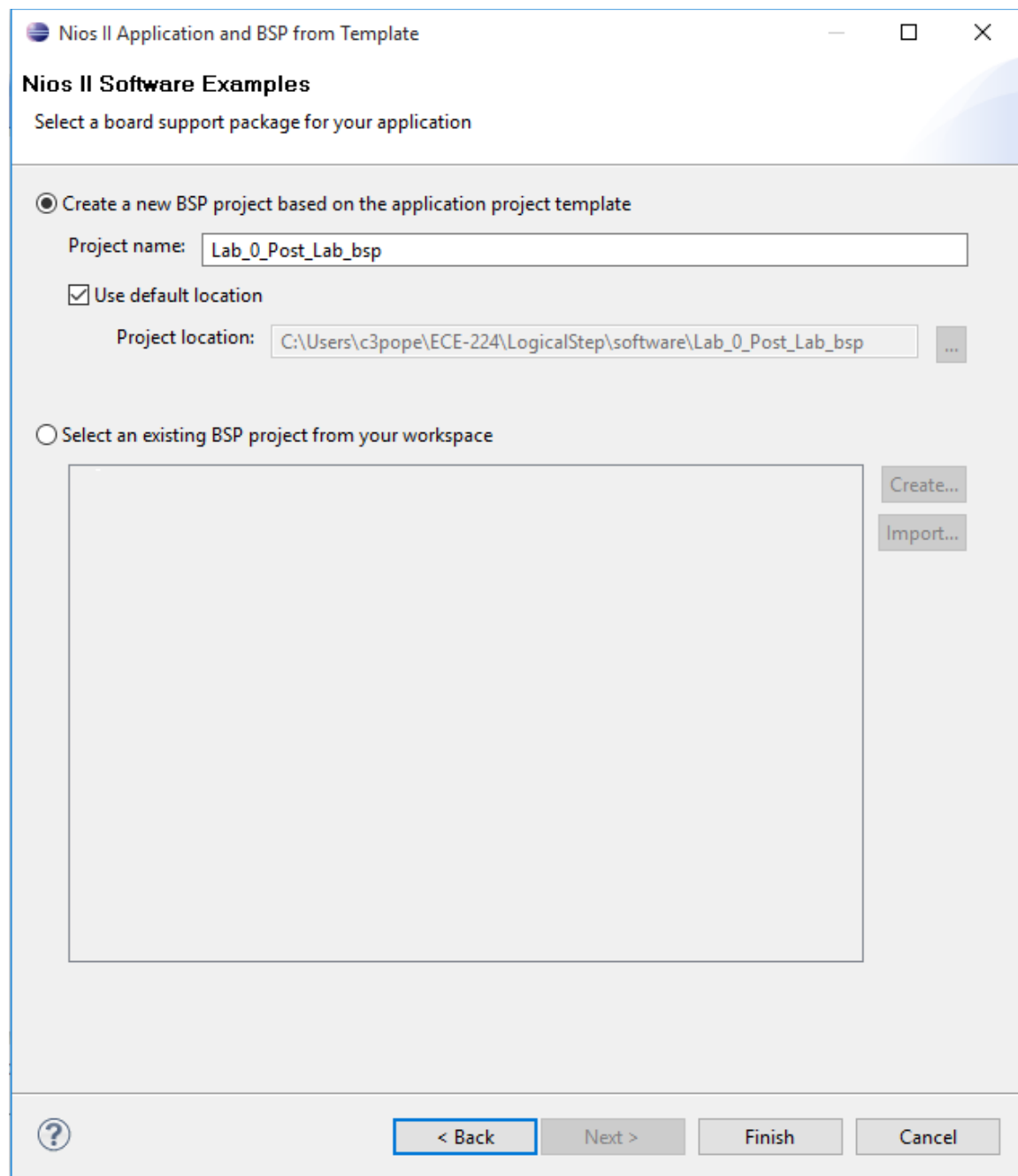


Figure 38: Creating a BSP Project to couple with the Lab_0_Post_Lab Project

8. Click Finish.

The NIOS/Eclipse IDE utility will now show up with the projects as shown in Figure 39. Notice how the Board_Diagnostics project (and its BSP project) are now “closed” but are still in the Project Explorer pane. Lab_0_Post_Lab and its related BSP are now the open and active projects.

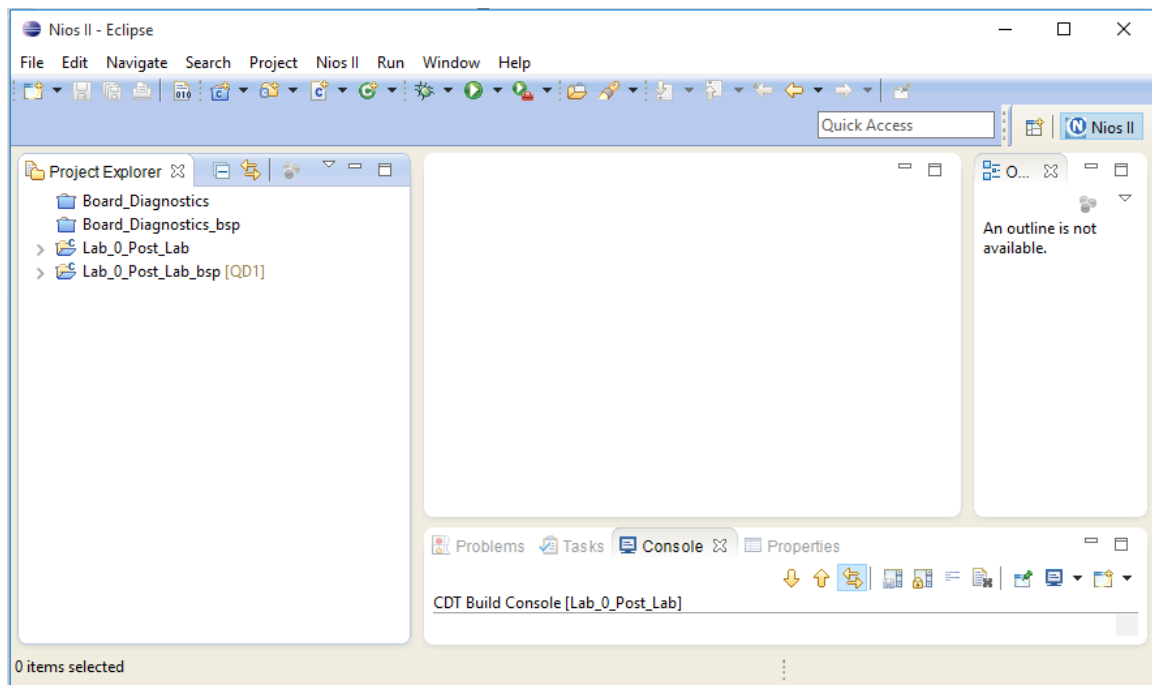


Figure 39: New Lab_0_Post_Lab Project is OPEN

To enter new c-code for this Lab_0_Post_Lab project expand on the "Lab_0_Post_Lab entry in the Project Pane. Then locate and select the "Hello_World.c" file and double-click it. Refer to Figure 40.

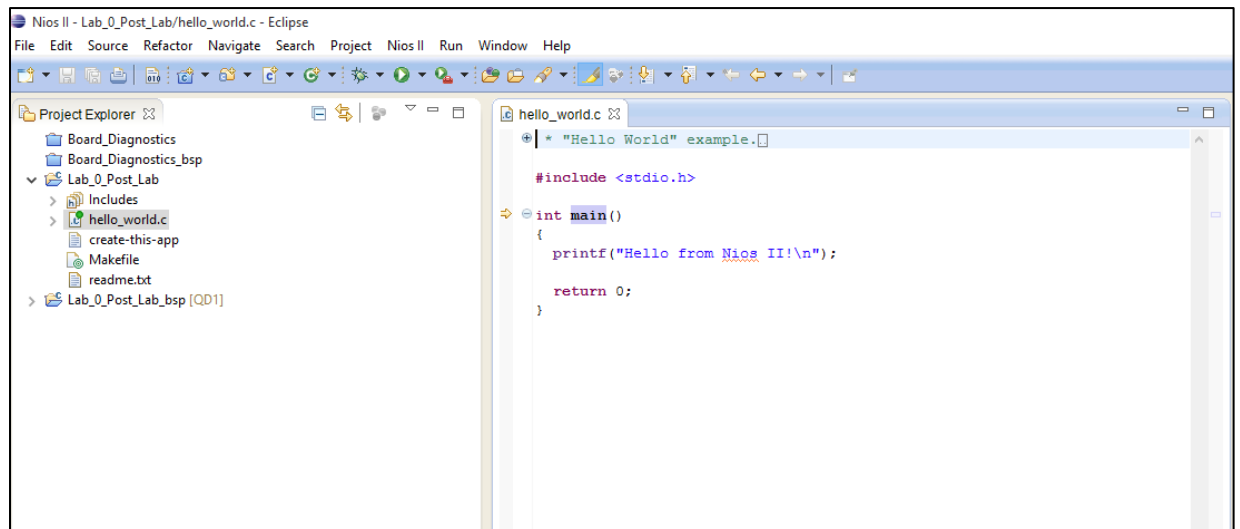


Figure 40: Accessing the Lab 0_Post_Lab "Hello_World" c-code

At the top of your “Hello_World”.c file, you’ll also need to add:

```
#include "system.h"
```

Altera has provided some handy macros to allow easy access to the PIO hardware included in your project. To use the macros you need to add the include statement:

```
#include "altera_avalon_pio_regs.h"
```

With this header file included you’ll be able to easily read and write to the PIO registers. For instance, to read the pushbuttons and save it to a variable called buttons you would write:

```
buttons = IORD(BUTTON_PIO_BASE,0);
```

Which calls the input/output read macro and passes the address and register offset as arguments. For the PIO cores a read or write to the base address (i.e., register offset of 0) will read or write the data from/to the parallel port.

Reading the pushbuttons using the code above stores a number that represents the state of the four push buttons simultaneously according to how their signals appear on the PIO. The push buttons are **active low**. If no push buttons are being pressed when the port is read, then the port would be b’1111 or 15 in decimal. A read while push button 0 is being pressed results in b’1110 and therefore the ‘buttons’ variable above would contain the number 14.

An **additional PIO** port that can be used in your Lab 0 Post Lab project is the one used for **switches**. Follow the same approach outlined above for the buttons to save another value to a different variable called “switches”. The name of the Base Address of the switch pio, from which you will be reading, is located in the system .h file. Note that the switches are active high.

As a complement to the read macro Altera provides the IOWR macro to enable writes to a PIO. If you would like to turn on every other LED on the logical step board you could write:

```
IOWR(LED_PIO_BASE,0,0xAA);
```

Where there are now three arguments, the hardware address, the register offset, and data to write to the parallel port. In this case the data is 0xAA or b’10101010 which turns on every other LED.

To understand what the register offset argument is, take a look at the table below that describes the assignment of the PIO that drives the seven segment display and the segment map in Figure 41. The port is 16 bits wide to drive all the LEDS but you’ll notice there are two totally

different mappings that can be used as shown in Table 20. This was custom designed hardware for the LogicalStep board to support both Altera's board diagnostic software at register offset of 0 and a more user friendly mapping at a register offset of 1. Both registers are part of the seven segment core, so they have the same base address but they have a different register offset.

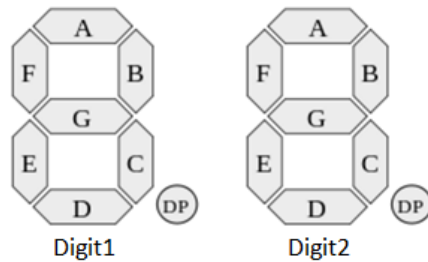


Figure 41: Segment Mapping for Dual 7 Segment Core

Table 20: Register Offsets for Dual 7 Segment Core

Bit Number →		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register Offset	Dir.	Digit 1								Digit 2							
0	W	\overline{DP}	\overline{A}	\overline{B}	\overline{C}	\overline{D}	\overline{E}	\overline{F}	\overline{G}	\overline{DP}	\overline{A}	\overline{B}	\overline{C}	\overline{D}	\overline{E}	\overline{F}	\overline{G}
1	W	DP	G	F	E	D	C	B	A	DP	G	F	E	D	C	B	A

Depending on which IP block in Qsys you are trying to access through your code there will be different registers available. For instance, a simple read from or write to a PIO uses the base address with a register offset of 0. However, there are a total of six registers in the PIO core four of which are both read and write. The details of the PIO core can be found in the Altera's PIO Core datasheet which provides a wealth of information about how it works and how to use it.

The majority of the IP cores in your project are provided by Altera and information on how to use them can be easily found online.



Tips and Tricks

It is worth looking up both the PIO Core and the Interval Timer Core datasheets from Altera to assist you while working in the lab. See [Altera PIO core](#) or [Altera Timer Core](#).

The EGM and Dual 7 Segment core were made in-house at the University of Waterloo, the details of these cores can be found in Appendix A.



Deep Dive

The NIOS II processor communicates to peripheral devices using the Avalon bus which was developed by Altera. The Avalon bus is in fact not a bus in the conventional sense as all bus connections are made in the FPGA fabric like all other hardware configured in the FPGA. The Avalon bus is better understood as a construct that defines the available interface between a master and slave device, which in this case is the NIOS II and a PIO in Qsys. The register offset is born out of a 32 bit address that is defined by the Avalon bus, an address often used to access different features within an Avalon slave device.



Take Away

Use the system.h file to determine the defined base address of a hardware core you are trying access. Use Altera's IORD and IOWR macros and the base address to move data to and from the IP cores in your Qsys system. Download IP core datasheets from Altera for more information about the register offsets and how to use their cores. See [Altera PIO core](#) or [Altera Timer Core](#). Check Appendix A for details about the EGM and Dual 7 Segment cores.

2.7.2 Lab 0 Post-Lab Exercises: Creating a Simple Read / Write Looping Routine

Test your understanding of reading and writing to the PIO cores by completing the following exercise:

In the application you just created, use the IORD and IOWR macros running in a looping fashion that reads values from the Push Buttons PIO core and the Switches PIO core. Make the Push-Button PIO core bits (3:0) appear on LED (3:0). Make Switches PIO Core bits (3:0) appear on LEDs (7:4).



Take Away

By completing the Lab 0 and Lab 0 Post Lab exercises you will have verified that your hardware works. You will have also seen how to read and write peripherals connected to your system by the PIO core.

3 Lab 1: Experimenting with Polling and Interrupts

This lab project will involve writing software that will be run by the NIOS II processor to service external event activity generated by the EGM module. To service these events the NIOS II processor will use two different methods: i) tight polling and ii) interrupts. While servicing these events the NIOS II processor will also be required to run background tasks.

3.1 Introduction

The objective of Lab 1 is to provide students with a deeper understanding of how a processor deals with external events. Often the main approaches used in dealing with external events are the “polling” method and the “interrupts” method. So for this lab you will be required to implement both of these methods (tight polling and interrupts) to respond to external events.

With both implementations you will be required to run experiments and collect data that will allow you to evaluate the benefits and drawbacks of each approach to handling external events. To facilitate experimental data collection you will have to write a program that can automate the experiments to test a large number of variables automatically and gather the required data.

3.2 Overview

In Lab 1 you will be required to use the NIOS II processor to service external events using tight polling and interrupts. The external events that you need to service are created by a hardware core in Qsys called the Event Generation Module (EGM). The EGM sends a pulse to the NIOS processor (the stimulus) and your software code needs to detect the stimulus and send a pulse back to the EGM (the response) as soon as possible. Every time the EGM sends a stimulus pulse the NIOS II must send a response pulse back as soon as possible. The EGM internally measures the NIOS’s response latency by measuring the time elapsed between the stimulus pulse and the response pulse – this latency can be used to help evaluate the difference between polling and interrupt based approaches to dealing with external events.

At the same time while the NIOS II is servicing these external events your software code will also be required to execute a background task function that simulates other work a processor could be required to do while waiting for an external event to happen. Therefore the differences between polling and interrupts can be evaluated experimentally by comparing both the response latency and the amount of background task work each approach can complete.

3.3 Lab 1 Software Project Setup

Create a software project for your Lab 1 code.

1. Close out (do not delete) any of the previous project components (Board_Diagnostics and Board_Diagnostics_bsp, Lab_0_Post_Lab, Lab_0_Post_Lab_ bsp) within the Project Explorer pane in the IDE tool. Do this by selecting each one and then right-clicking on those items and select “Close Project”.
2. From the top menu select File > New > NIOS II Application and BSP from Template
3. Select your .sopcinfo file as you did for the earlier lab projects (QD1.sopcinfo)
4. Give your project a name with no spaces (“Lab_1”). Leave the “Use default location box selected.
5. Select the Hello World project from the Templates list as shown in Figure 42.

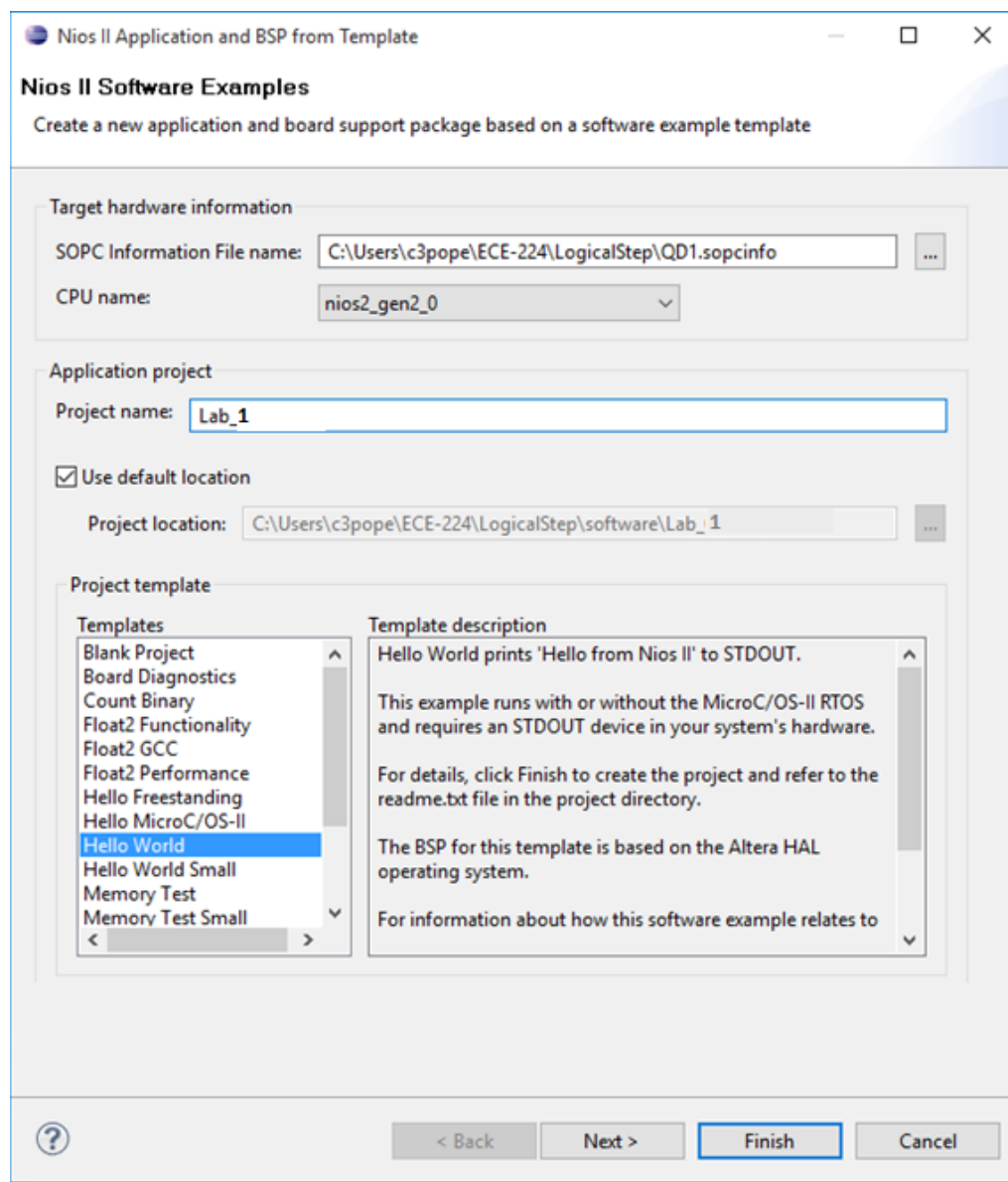


Figure 42: Configuring a New Application from a Template for Lab_1

6. Click Next
7. Similar to what was done for Lab_0_Post_Lab earlier create a new BSP project to be coupled with the Lab_1 project click on the “Select a new BSP project based on the application project template”. Let the project BSP file default to the Lab_1_bsp value. Leave the “Use default location” box selected.
8. Click Finish

Now that we have the project setup there are two changes to the BSP settings that need to be made to ensure the data produced for Lab 1 is as accurate as possible. These changes will reduce operational “overhead” during the running of your Lab1 experiments.

1. Right click on the “Lab_1_bsp [logical step]” and select NIOS II -> BSP Editor...
2. When the BSP Editor opens, it should be on the main tab by default. Change the “sys_clk_timer” to “none” as shown in Figure 43.

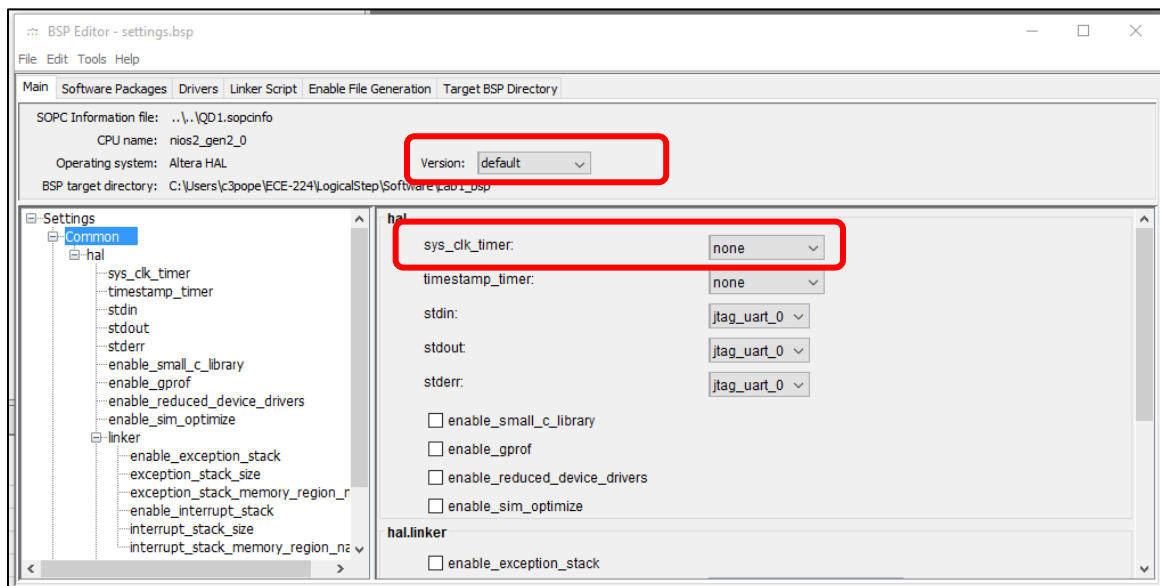


Figure 43: Changing the sys_clk_timer Settings in the BSP

Now referring to Figure 44 select the Drivers tab. Check the box to “enable_small_driver” for the “altera_avalon_jtag_uart_driver”. This ensures the printf function won't be interrupting your tests to spool out characters to the console (the small driver halts the program until all characters are printed).

3. Click Generate

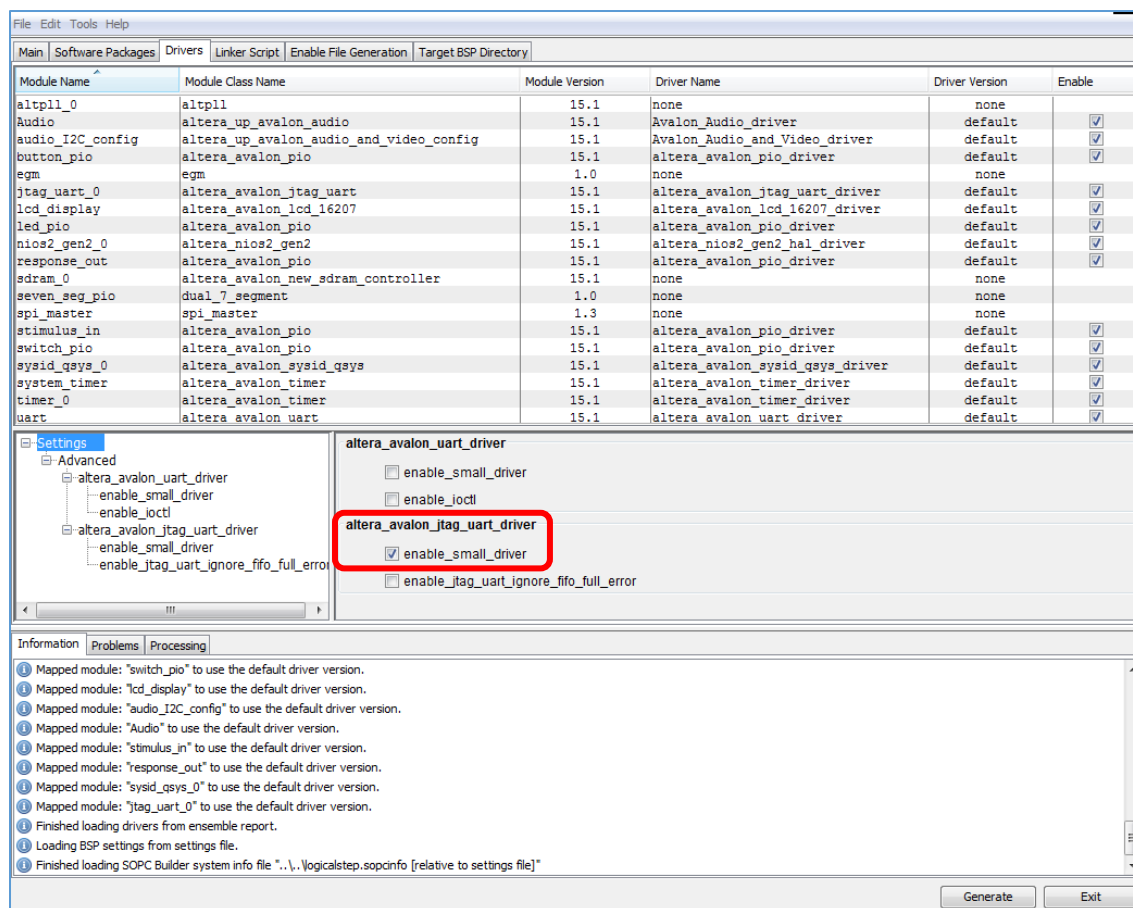


Figure 44: Enabling the Small Driver for the JTAG UART

3.4 EGM Module and its Use in Lab 1

The EGM IP core is a custom core developed at the University of Waterloo for this lab. When you built the hardware for Lab 0 in Qsys you included the EGM core and two PIO cores named Stimulus In and Response Out. In the top file you used two 'wires' to connect the EGM stimulus and response ports to the related PIO cores as shown in Figure 45.

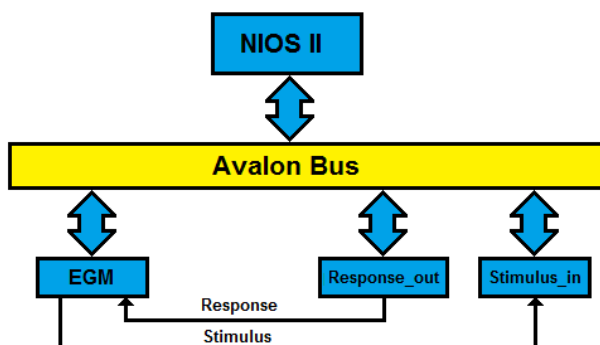


Figure 45: EGM and NIOS Topology

Note that the NIOS processor can communicate with all three cores over the Avalon bus. To gather data for the lab report you will be required to implement and test both approaches (polling and interrupts) on the “Stimulus In” PIO core. When your code detects the stimulus you will then simply send a response pulse using the “Response Out” PIO core.

To start the experiment you will have to configure the EGM to send out the appropriate type of pulses and then enable the EGM **for each run of the test**. Once the EGM is enabled it will send out a stream of pulses for a specific and consistent period of time and then it will stop the pulses automatically. Once the EGM stops, your software code will be required to access the EGM and gather the average response latency and total missed pulses for that run of the test and then it is to disable the module to reset it. The parameters of the EGM will then be initialized to a new set of values and then the NEXT test will begin.

To configure the EGM the user must set a ‘pulse width’ and ‘period’ value in the respective registers of the EGM prior to enabling the EGM to run the test. The EGM has an internal counter that is incremented by the clock. When the EGM is enabled it will set the stimulus pulse high until the counter value matches the user defined ‘pulse width’ value at which point the EGM will set the pulse low. The stimulus is kept low until the internal EGM counter equals the user defined ‘period’ value at which point the counter is reset and the EGM cycle restarts by setting the stimulus pulse high again as shown in Figure 46. For Lab1 the Stimulus pulses are to be maintained with a 50% duty cycle (i.e.: pulse_width = 50% of period).

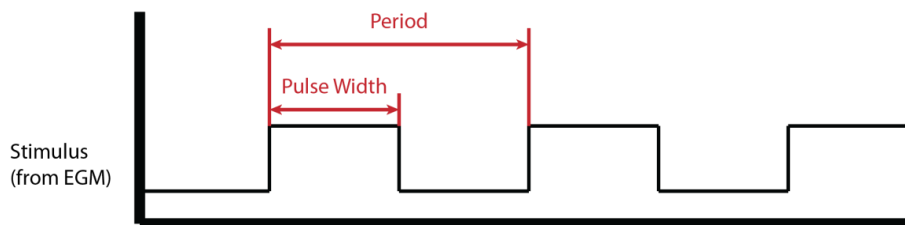


Figure 46: EGM Stimulus Pulse-train Settings

The stimulus cycle continues to repeat until the EGM is finished the test and it stops pulsing. The user must query the EGM’s ‘busy’ register during the test to determine when the EGM is finished. A value of 0 in the EGM’s ‘busy’ register indicates the EGM is finished the test. When the test is complete the user must query the ‘missed pulses’ and ‘average latency’ registers to record the results for that test before setting the **EGM enable low**. The register offset values for interfacing with the EGM over the Avalon bus are described in Appendix A.

For every cycle the EGM measures the response latency of your code implementation by counting the elapsed clock cycles between the leading edge of the stimulus pulse (sent by the EGM to the NIOS) to the leading edge of the response pulse (sent by your software code in the NIOS back to the EGM) as shown in Figure 47. The latency will vary across the EGM pulse cycles due to the “loop time” taken by your software and the “cycle time” of the EGM parameter-driven operations.

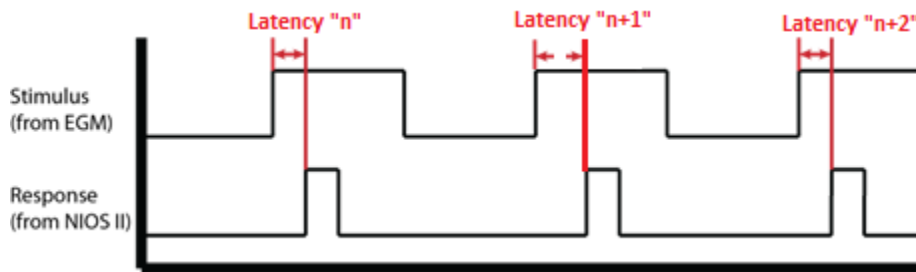


Figure 47: EGM Latency Measurement

Every latency measurement gathered by the EGM in a test is used to generate an exponentially weighted moving average of the response latency that is available to the user at the end of the test.



Watch out! _____

While the trailing edge of the pulses are ignored by the EGM’s latency measurement, the user must ensure the response pulse is set to low before the EGM sends the next stimulus pulse to avoid a missed pulse being counted. Thus, for each stimulus cycle, the response must be set to High and to Low before the next rising edge of the Stimulus is sent out by the EGM otherwise a Missed Pulse will be counted.

3.5 Lab 1 Code Development for Executing a Test

For Lab 1 the first step is to write C-code to execute a test using the EGM. Each test must meet several objectives in the following order of importance,

1. Missed Pulses: tests that accumulate some pulses being missed are considered as invalid tests. Under certain circumstances having tests that result in missed pulses will be unavoidable. For those tests record the test data but graph the data in a different color to indicate the test results included missing pulses.
2. Shortest Latency: The code should be written to achieve the lowest average response latency.

3. Highest Background Task Completion: The code should be written to complete the most amount of background tasks during the test.



Tips and Tricks

Since shortest latency is more important than completing the most background tasks the focus must be maintaining the lowest latency first while completing the highest amount of background tasks second. This concept is most critical during tight polling code development.

3.5.1 *Recommended Approach to Interrupt Test Development*

The interrupt based stimulus response is easier to complete than the tight polling approach as the background task call can simply be executed in a while loop that is interrupted each time a stimulus pulse is received. It is recommended to use a simple approach to a response pulse within the 'stimulus_in' ISR (i.e.: set 'response_out' high then immediately set the 'response_out' low within the ISR code). Refer to Appendix B for more details.



Tips and Tricks

It is recommended to implement the interrupt approach before the tight polling approach first. Once this approach is working then start on the tight polling approach.



Watch out!

Regardless of whether you are doing tight polling or interrupts, to record good data you **must** ensure the background tasks are not being run after the EGM has disabled itself. This means that **before every new background task call you must ensure the EGM is still active** and by extension that the test is still running. If you don't do this your results will be incorrect.

3.5.2 *A Possible but Incorrect Approach to Tight Polling Test Development*

Completing the tight polling test is more challenging than interrupts because of the requirement to do background work while maintaining a low amount of response latency.

The first way to achieve both objectives is to poll for the edge of the stimulus and once the stimulus is detected, run the background task a given number of times before starting tight polling again to detect the next stimulus pulse as shown in Figure 48.

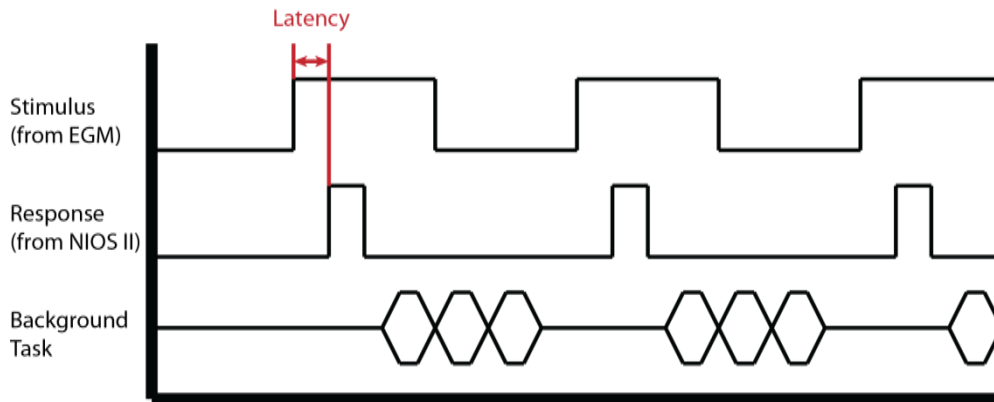


Figure 48: Running a “given number” of Background Tasks during Tight Polling

Note how in figure the background task is run three times after the stimulus is responded by the Nios II. After the three background tasks are completed, the NIOS resumes tight polling to detect the next stimulus pulse with a minimum response latency while at the same time completing the most background tasks. Note that you **must** ensure the EGM is still enabled anytime you intend to call another background task (otherwise your results will be incorrect).

The problem with this scheme is that it is impossible to know *a priori* how many background tasks can be executed and completed before the next stimulus pulse arrives from the EGM.

3.5.3 Recommended Approach to Tight Polling Test Development

Therefore a “characterization” approach **must** be used during the first cycle of the test to automatically determine the number of background tasks that can safely fit within each period. With the first cycle of the test being used for a “characterization cycle” every subsequent cycle can be executed with a safe number of background tasks being completed and also to ensure that the minimum amount of cycle response latency is being achieved. An example of how this would work is shown in Figure 49.

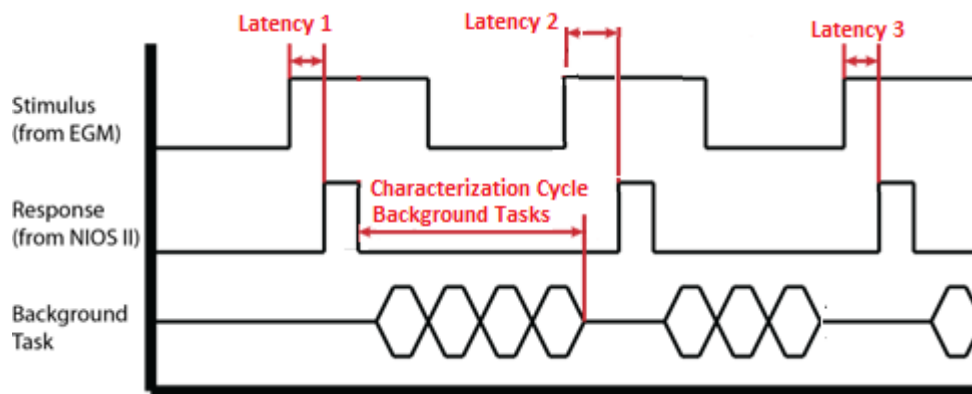


Figure 49: Characterizing Background Tasks during Tight Polling

For the first cycle the NIOS would use tight polling to respond to the first stimulus pulse. As soon as the response pulse is sent the NIOS will call a background task. When the background task is complete the NIOS will check to see if a new stimulus pulse is received. As long as no new stimulus pulse is detected the NIOS continues to call another background task. Eventually the NIOS will detect a new stimulus pulse, in the figure above it occurred after four background task calls. The characterization code then determines that three calls to the background task are safe for the remaining Stimulus pulse cycles of the test.



Deep Dive

The approach to tight polling used in Lab 1 is a hybrid approach as the processor is still able to get background tasks done during part of the cycle before it starts tight polling for the stimulus pulse. This approach results in latency values identical to normal tight polling but allows background tasks to be completed as well by using a characterization approach to determine if background tasks can be safely completed between stimulus pulses.



Tips and Tricks

During the characterization cycle above there will be extra latency recorded for the second cycle because of the extra background task. However, because the average latency is calculated using an exponentially weighted moving average it will have little to no effect on the final reported average latency.

You are required to write software code that automatically characterizes the number of background tasks that can be called for each cycle while maintaining the shortest possible response latency. The software code must have a way to detect each new stimulus pulse as

distinct from the current stimulus pulse. **DON'T USE THE EDGE-CAPTURE REGISTER IN THE STIMULUS PIO CORE.** You must use the “raw” Stimulus input signal available in the Data register of that core. This will allow your software-driven responses to get closer to the actual asserting edge of the Stimulus pulse.



Watch out! —————

If you use a while loop to do tight polling one of the conditions of the while statement must be to test that the EGM is still running – if not your code will hang at what appears to be random times in an experiment because the tight polling is waiting for the next stimulus pulse after the EGM has been disabled.



Tips and Tricks —————

During the cycles within the test you will want to be checking the EGM busy register to determine when the EGM is disabled and the test is complete.

3.6 Lab1 Code Development for Executing the Experiments

An experiment, in the context of Lab 1, is made up of a series of EGM tests. Each EGM test consists of one set of parameter values to configure the EGM for generating a repeated and consistent pulse train for a fixed duration of time. The duration of the EGM test is not revealed to students.

The ‘experiment code’ will run the ‘test code’ over and over with different parameter values for the period and pulse width until the entire range of parameter values has been covered..

Between each test code run the experiment code will read out the results from the EGM registers and then send them to the “stdout” port along with the parameters used for that test.

(It is recommended to print the test data to the console using CSV format so you can easily import the data into excel). The latency and background task results should cover all the required period values as specified in the Lab 1 Objectives and Deliverables section.

You can write one experiment code for testing tight polling and a second experiment code for testing interrupt response – however they must both be in the same source file using #ifdef

statements. The automatic characterization for the tight polling test is required because the experiment code will run the test code thousands of times to gather the required data.

3.7 Lab 1 Objectives and Deliverables

Using the information presented from the last chapter as a background to this lab, you will need write software c-code to complete two experiments. One experiment for interrupts and one for tight polling with both implementations located in the same source file (use `#ifdef` statements to select between the different implementations at compile time). Each experiment must test the full range of EGM stimulus pulse period settings from a period of 2 to a period 5000 in steps of 2 clock cycles (i.e., 2, 4, 6, 8...). You should always maintain a duty cycle of 50% (i.e., if you run a test with the EGM period setting at 400 then the pulse width setting should be 200). The experiment code can be a 'for' loop that increments through different EGM period settings each time the test is run. To gather the data you can use comma delimited `printf()` statements to report the period, duty cycle, total background tasks run, average latency, missed pulses out to the console. From there you can move it into excel to graph it for analysis.

3.7.1 Lab 1 Demo

You should be ready to demonstrate your implementation by the start of the third lab session. For the demo, you will be expected to explain how you implemented your experiments. You will also be asked to run your code. Both partners are expected to be able to answer questions regarding the implementation of your project. The marking for the Lab 1 Demo will be as follows:

1 mark – for setting up an experiment to run in ONE of two modes (interrupt or Tight-Polling) and having the mode and the test parameters/results headings should be displayed on the NIOS2 Terminal Console before the test starts. The mode choice will be made by the TA. The experiment run should WAIT for a PBO press before proceeding.

1 mark- for running the chosen experiment and having the test data appear on the NIOS2 Terminal Console. The individual test parameter and result values should only be reported to the NIOS2 Terminal Console at the end of each test run.

2 marks – for your c-code design and your reasons for your c-code implementation.

3.7.2 **Lab 1 Experimental Results for the Lab1 Report**

For Lab 1 you are required to test tight polling and interrupts as discussed previously in this section by gathering data that relates the EGM period setting to both the background tasks completed and the average latency. Therefore, you should have four datasets gathered to write your report. It is recommended that when you are gathering and analysing data in the lab you ask the lab instructor or TA's for guidance on whether your graphed data looks correct before starting the report.

3.7.3 **Lab 1 Report**

Your report must **not** include a full dataset or source code; your report must include the four graphs of the collected data as described below. Your marks for the report will be based on your ability to recognize important details in the presented graphs and your ability to correctly explain the root cause of those details. For each graph presented it is important to call attention to any details in the graphed data (overall trend, slopes, curves, discontinuities etc.) and provide a concise description of the root cause for the trend in the data. This will require a thorough understanding of how your code works and how the EGM works within your approach to tight polling or interrupts so that you can accurately explain the trends in your graphed data.

Your report should include four sections that cover the following details:

- Present your background task completion data from the tight polling experiment in a plot ('Background tasks vs EGM period' from a period setting of 2 to 1500 clock cycles) and explain the results presented with the aid of logic waveforms. You must point out sections in the graph and explain the root cause relating to: missing pulses, the overall trend-line, discontinuities and the negative sloped curve between discontinuities. **(20%)**
- Present your background tasks completion data from the interrupt experiment in a plot ('Background tasks vs EGM period' from a period setting of 2 to 1500 clock cycles) and explain the results presented. You must point out sections in the graph and explain the root cause relating to: missing pulses, the overall trend-line, background task efficiency. **(20%)**
- Present your latency data from **both** the tight polling and interrupt experiments in a single plot (Latency vs EGM period from a period of 2-1500 clock cycles) and draw comparison conclusions about the different approaches using the data presented. You must point out sections in the graph and explain the differences observed – there are two key differences we expect you to explain. **(20%)**
- Present your background tasks data from **both** the tight polling and interrupt experiments in a single plot (Background tasks vs EGM period from a period of 2-5000 clock cycles) and draw comparison conclusions about the different approaches using the

data presented. You must point out sections in the graph and explain the differences observed – there are two key differences we expect you to explain. **(20%)**

- You will also be graded on the quality of your presentation. Areas of interest in the graphs must be clearly labeled on the graph and the description presented under the relevant label in the text. The report presentation should be consistent with a formal lab report for full marks in presentation. The report must not exceed 5 pages in length, one page per graph and explanation, plus one title page – minimum 11 point font **(20%)**.



Tips and Tricks

When discussing data in the report be sure to pay careful attention to discontinuities, slopes, curves etc. in the graphs of recorded data.



Watch out!

A written explanation of what the graph looks like will receive no marks. Marks will only be awarded for specific details in the graph (dominant details) that are called out and the underlying root cause is explained with reference to the code or function of the experiment.

4 Lab 2 Based Practicum: Engineering Design Services Case Study

The second lab for the course is considerably different than what you've experienced in the past for lab work. The objective of this lab is to build upon what you've learned in lab 1 while getting a taste of a real-world engineering experience. This lab has minimal step-by-step directions to guide you to completion. You will have to use the knowledge of the tools and coding experience gained from lab 1 to complete lab 2. It is suggested that you split the project development into 3 parts:

- i) Lab2_LOOP_AUDIO
- ii) Lab2_CLI_Testing
- iii) Lab2_WAVE_PLAYER

For each of the above parts you can start a project in the NIOS II SBT tools like was done in Lab1 ("Hello_World"). Remember that you can only have ONE project and its associated BSP project active in the NIOSII SBT Panel frame at one time. All others must be closed. As you move through the different project phases you can turn on or off the project/project BSP pairs as you require. For each of the above 3 phases use the BSP Editor (as was done in Lab1) to set the altera_avalon_jtag_uart option to use the "small C driver". Leave the sys_clk_timer setting as System Timer.

There are three deliverables for this lab: your time sheet estimates, a demonstration of your completed project and a reflective report. The demo will be primarily graded according to the quality of the solution. To achieve full marks the solution presented in the demo must satisfy all of the specified requirements.

Each group is expected to thoroughly review the supplied materials **prior** to starting development in the lab and generate an effort estimate for the project that includes both a breakdown of required tasks and how many hours each task is expected to take. During the project you must accurately record how long each task actually takes and present the supplied timesheet for review as part of the reflective report.

4.1 A Message from the LogicalStep Company's Solution Architect

Engineering Team,

We received a sales call this week from a customer who is looking to have a music player built. They haven't given us too many details about the final product, as it is on a need-to-know basis. They will be handling final integration, and we are only responsible for the electronic functionality by supplying them with a board that meets their needs.

Senior management reviewed the sales proposal and determined that our LogicalStep board would fit the customer's hardware requirements as-is. This means we can make a reasonable profit on the job as only the software portion will require design cycles. We need your team to put together an effort estimate in hours for the software portion of the music player design and forward it to the sales team to use in preparing a quote for the customer. Before you start any work on the project, take some time to review the enclosed materials and get an idea of what work will be required and how long it will take you to complete each portion. Give your best estimate – don't build in any extra time margin as the sales team will add in margin on top of your best guess. As always, you will need to keep an accurate record of the time required to complete the project using the included time-sheet. Please be careful to mark down all time spent to an accuracy of 15 minutes so we can do a project evaluation exercise to review the accuracy of the quoted effort.

The detailed customer requirements for the project are attached, but I'll give you an overview. The customer wants a digital audio player to integrate into a product they are developing. It fits well with the LogicalStep board because they want a device that can play stereo wav files off a MicroSD card and send the output to a standard headphone jack. Aside from this basic functionality, it needs a user interface that displays the file name on an LCD and has buttons to start or stop the song as well as search through tracks as described in the specifications.

During the call I did some digging to see what materials we could leverage to expedite this job and found some excellent resources. The LogicalStep hardware uses the Altera University Program Audio IP core in the FPGA design, so I pulled the audio core document from the Altera site and included it in the materials on LEARN. The last page of the document ([Figure 2](#)) has example code to read from the audio input and then write to the audio output. This is a good place to start with the audio development. I've also included some additional notes I found from a previous project.

In the past, when we used the MicroSD card we started a development using the FatFS module to do the heavy lifting related to the SD card file system. This FatFS module is based on the FAT32 format and will be used for this project as well. I've included this utility in the materials for the project and I also added a command line interface (CLI) project that you can use through the terminal to mount the drive, read the files and explore the FatFS module. I've added an example with some of the commands you'll need to use in the CLI.

Don't forget that when you are accessing the audio data from the SD card with the FatFS module it is in a sequential "byte by byte" fashion. But the .wav file has the audio data packed as 16 bits for the left channel then 16 bits for the right channel, making up one stereo sample. So essentially you'll need to take the first two bytes and make a 16 bit number out of it and pass

it to left FIFO then take the next two bytes make a 16 bit number out of it and pass it to the right FIFO and repeat.

Once you have audio playing, you can start on the final design which includes the user interface. You'll need to extract the required FatFS library calls from the CLI project and put them in your final project so that the player works using only by the buttons and LCD screen. The details of the user interface can be found in the client's specifications document.

Again the hardware should be fine "as-is" and all the required files for this project are zipped up on LEARN. The Altera/Eclipse NIOS II development platform will be used for this project. I would recommend completing the project development in 3 phases as outlined below. For each of these phases create a separate NIOS II software project in the Altera Eclipse utility.

Phase i) Lab2 Loop Audio:

- See the instructions in APPENDIX F: Lab2 Hardware Testing resource; - make sure that your audio pipeline hardware is fully functional before proceeding to the next step.

Phase ii) Lab2 CLI Testing:

- When the audio pipeline is working move on to this second Lab2 NIOS II project phase. Download from LEARN the FatFS CLI and support files and insert them into this software project directory. Copy the "main.c.txt" code into the hello_world.c file. See the instructions in APPENDIX G: Lab2 FatFS Command Line Interface – Tips and Tricks resource. Develop your understanding of how the CLI uses FatFS calls. Use the Altera/Eclipse Debug tools with single-step and or breakpoint operations as an aid. There are also good general background references for FatFS details on the web: (See: http://elm-chan.org/docs/fat_e.html , http://elm-chan.org/fsw/ff/00index_e.html, and http://elm-chan.org/docs/rc/fat_map.png)

To play the audio from the SD Card to the audio interface copy the audio code from the Lab2_LOOP_AUDIO/hello_world.c file into the new Lab2_CLI_Testing/hello_world.c file in the 'File Play' (fp is currently blank) section. You can learn to use the CLI to mount the drive, initialize the FatFS and then open the file and play the given number of bytes for the audio wav file. Your task for this phase is to interface the FatFS Buffer to the Audio interface output FIFO's. Determine the best Buffer size (the original length definition is 8192 bytes) and the ordering of the SD card audio byte data into 16 bit word samples to get the audio wav files to play without distortion. Monitor and make sure that the Audio output FIFO's don't overflow or underflow.

Phase iii) Lab2 WavePlayer:

Once you can play audio files using the CLI without distortion, start development of the final project phase code using the Lab2_WAVE_PLAYER / hello_world.c file as a starting point and then extracting some portions of the FatFS calls (f_mount, f_opendir, ... etc) contained in the CLI code from the previous project phase. You will have to make sure that you are using pointers etc. properly.

There are other certain functions in the CLI are only to report your FATFS command results only. These are like put_rc(), xput() etc. Others like xatoi() are used to interpret keyboard input.

You should be able to get this project phase completed in small increments without having to learn the fine details of the FAT structure. (Hint: use the FILINFO records).

Then you must create the following functions:

- a. isWav Code: Write a function: `int isWav(char *filename)` which accepts a pointer to a filename string and determines if the file is a wav file.
- b. Song Index: Write code that is based off of the 'File List' command code that checks each filename and if it is a .wav file stores the filename and file size into respective `char filename[20][20]` and `unsigned long fileSize[20]` arrays. You may want to include `string.h` in your project so you can use `strcpy()` to make things easier.
- c. LCD Display: Write code that displays the filename on the LCD and allows you to cycle through the different .wav files using the push buttons. Refer to previous labs and to the Embedded Peripheral Datasheet on LEARN in the Lab Datasheets section for the LCD Controller.

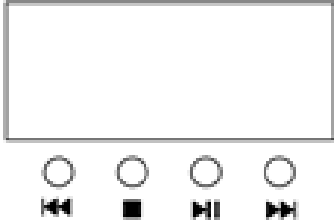
At this point you can add the user interface requirements for the product application. Complete the rest of the player functionality as described in the attached specifications.

NOTE: the Push buttons must all be "de-bounced" by you.

This should be a fun project and I'm really interested to see what you guys come up with. Good luck!

Jim Miller
Solution Architect
LogicalStep

4.2 Client Provided Design Specifications for Audio Player Product

Specifications	
Must Include	<ul style="list-style-type: none"> - The audio player must support .wav format audio files containing stereo audio with 16 bit audio samples. - The audio must be read from a MicroSD card - The audio output uses a standard head phone jack - The hardware must include an LCD display that indicates the track number (<u>as enumerated on the SD card</u>) and the track title (filename) on the first line of the LCD display. - <u>The player must only display track titles (the filename) that are actual wav files indicated by the .wav extension.</u> - The audio played must be free of distortion - User interface must function as described
User Interface/	- Button arrangement must be as shown below
Button	
Functionality	
	<ul style="list-style-type: none"> - The <u>Play/Pause</u> button plays the currently selected track, as currently displayed on the LCD, if the player is stopped. The Play/Pause button pauses the track if the track is already playing. The Play/Pause button will continue to select the current track position if the player is paused and the audio must stop. This button must be de-bounced. - The <u>Stop</u> button stops a currently playing track and resets the track position to the beginning of the track. If a track is paused, the Stop button resets the track position to the beginning of the track. This button must be de-bounced. - The <u>Previous</u> button searches backwards through the tracks on the SD card, one track each time the button is pressed, and displays the current track name on the LCD if the button is pressed. Searching farther backward past the first track in the list should cause the last track to be displayed. If the button is pressed while a track is not playing (stopped or paused) then the player just moves to the start of the new track. If the button is pressed <u>while a track is playing</u> then the playing continues but it begins at the start of the new track. This button must be de-bounced. - The <u>Next</u> button searches forwards through the tracks on the SD card (one track each time the button is pressed), and displays the newly searched track name on the LCD. Searching farther forward past the last track in the list should cause the first track to be displayed. If the button is pressed while a track is not playing (stopped or paused) then the player just moves to the start of the new track. If the button is pressed <u>while a track is playing</u> then the playing continues but it begins at the start of the new track. This button must be de-bounced.

4.3 Timesheet

One of the challenges of engineering project management is the ability to accurately quote the time required to complete a job. One way to improve your estimation skills is to carefully consider a job before you start, record your estimated effort and then see how it compares to the actual development time. One great approach to improving estimates is to break a large project into distinct tasks, then estimate the required effort for each task individually. Combining these estimates gives an overall total.

Carefully read through the documentation provided here and on Learn then review the provided code to get an idea of the work required to complete the project. Fill in the timesheet below with your estimates of the required effort. Your estimates will be submitted as your pre-lab deliverable as described in Section 4.4.1.

Keep an accurate account of the time required for each task as you complete the project to see how it compares to your original estimate. If you run into a task that takes significantly longer than your estimated time, make a note of the reason or problem you ran into. This will help you with your report as described in Section 4.4.3.

The time sheet below has been prefilled with tasks from the Jim Miller's recommended approach to completing the job. Extra space has been provided in the table for you to add tasks required to complete the rest of the project. Brainstorm what other tasks you could add to subdivide the task of completing the user interface and quote the effort for each task.

Use this timesheet to record your hours on the project according to assigned task. Combine both group member's hours and record hours to an accuracy of 15 minutes (0.25 hrs). Example: if both group members work together for 1 hour and 15 minutes record 2.5hrs. Use the "Estimated hours" column to record what you think the effort will be. Then track your actual development time by using one of the adjacent columns for each day you work on the lab.

✂

Date (MM/DD)	Estimated Hours	/	/	/	/	/	/	Total
Project Setup								
Phase 1: Audio Development								
Phase 2: FatFS CLI Testing								
Phase 2: Playing ".wav" Audio								
Phase 3: isWav Code								
Phase 3: Song Index								
Phase 3: LCD Display								
Phase 3: User Interface								
Phase 3: Wave Player Operation								

4.4 Lab 2 Objectives and Deliverables

There are three deliverables for Lab 2.

4.4.1 *Prelab Timesheet Exercise*

Credit for completing the prelab timesheet estimates will be determined during the first session of Lab 2. Defensible estimates of required work for each task area are required as part of your reflective report submission. A printout or electronic copy of the timesheet must be used to record hours during lab sessions. Your estimates must be uploaded to Learn before the end of your first lab session to receive credit, which will be awarded as part of your reflective report grade.

4.4.2 *Demonstration*

Marks will be awarded for meeting each of the customer defined design criteria. Marks will not be awarded if the listed design criteria are not completed as described in the requirements section. During your demo the TA will review your code, ask questions about your implementation and test your audio player on the board.

4.4.3 *Reflection Report*

After completing your demonstration, reflect on your experience with this project by answering the following questions. Your report should be a maximum of 2 pages and does not need to include a cover page.

1. How accurate were your time estimates?
 - a. How did you arrive at your estimated times?
 - b. What time requirements did you estimate well?
 - c. What time requirements were poorly estimated?
2. What would you do differently if you were to complete a similar project in the future?

Appendix A: Custom IP Cores

Dual 7 Segment Core Register Description

The Dual 7 Segment core has two register offsets, 0 and 1. The register offset 0 is the default offset and is mainly used by the NIOS II Build Tools Board Diagnostics template to drive the dual 7 segment display. Note the position of the LED segments for seven-segment display in Figure 50.

Users wishing to write their own code to interface with the seven segments on the LogicalStep board are encouraged to use the more straight forward register 2 offset in Table 21. As an example, to display '7.0' on the seven segment display using register offset 1 you would write b'1000011100111111 or 0x873F to the register. However both offsets are detailed in the table below.

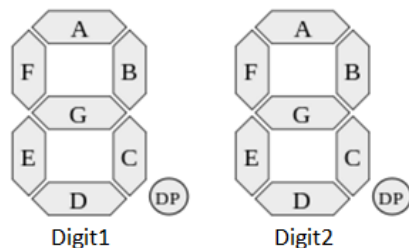


Figure 50: Segment Mapping for Dual 7 Segment Core

Table 21: Register Offsets for Dual 7 Segment Core

Bit Number →		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register Offset	Dir.	Digit 1								Digit 2							
0	W	\overline{DP}	\overline{A}	\overline{B}	\overline{C}	\overline{D}	\overline{E}	\overline{F}	\overline{G}	\overline{DP}	\overline{A}	\overline{B}	\overline{C}	\overline{D}	\overline{E}	\overline{F}	\overline{G}
1	W	DP	G	F	E	D	C	B	A	DP	G	F	E	D	C	B	A

EGM Core Register Description

The EGM core has 6 register offsets that can be accessed using Altera's IORD and IOWR macros.

Table 22 shows the register map of the 6 register offsets, the direction of the register and which bits are applicable for that register. The typical approach to using the EGM core is to first set the period and pulse width registers to define the type of stimulus pulse you want the EGM to send. Figure 51 shows the effect of setting the period and pulse width settings on the stimulus pulse. Note that the period setting (units measured in clock cycles) defines the number of clock cycles between rising edges of the stimulus pulse train and the pulse width setting (units

measured in clock cycles) defines the number of clock cycles that the stimulus pulse stays high. The pulse width setting should always be shorter than the period setting for predictable operation of the EGM.

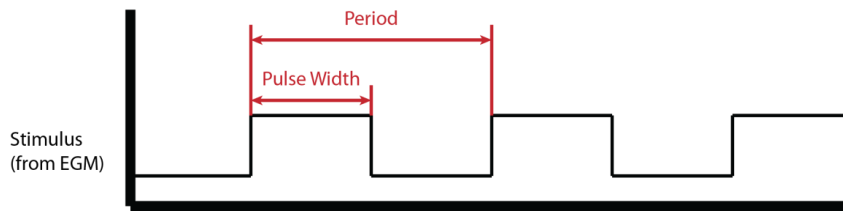


Figure 51: EGM stimulus pulse

Once the period and pulse width has been set the user can start the EGM by setting the enable bit. The EGM will send the defined pulse train for a predefined interval during which the average response pulse latency is measured using an exponentially weighted moving average filter and missed pulses are incremented every time a stimulus pulse does not get a response. There is no latency penalty for missed pulses – missed pulses are completely ignored by the latency tracker.

Since the EGM runs for a fixed interval once enabled the user must always sample the busy register to determine when the busy bit is cleared (the EGM stimulus pulse train is complete). Once the busy bit is cleared the user can then read the final latency and missed pulse values. The user **must** also clear the enable bit before being able to restart the EGM for another test.



Watch out! —

If you don't disable the EGM after the final run the hardware will not work the next time you try to use it. Enabling the EGM when it is already enabled will not start an EGM cycle.

Table 22: Register Offsets for EGM Core

Register	Dir.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0:Enable (BASE)	W																X
1:Busy	R																X
2:Period	W	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
3:Pulse Width	W	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
4:Latency	R	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
5:Missed	R	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Appendix B: NIOS Interrupts

Interrupts have many useful purposes, in embedded applications the concept of interrupt hardware provides the ability for the processor to continue doing useful work while waiting for an external event to happen. This is in contrast with polling which monopolizes the processor while waiting for an external event.

As is always the case with soft core processors on FPGAs there is a both a hardware component and software component to consider. Since the hardware is flexible and a function of design we must first consider the hardware design to determine the requirements of the software. In the case of handling interrupts, the code you will write that utilizes interrupts needs to consider the hardware you have designed. So we'll discuss the hardware first.

Hardware Considerations

When you define a NIOS system in Qsys there is the ability to add signals specifically designed to interrupt the processor via the NIOS 'Interrupt Receiver' which is standard on the NIOS processor. Should the NIOS be interrupted by a core the NIOS can query the core over the Avalon bus to determine the reason for the interrupt in more complex cores such as the PIO.



Deep Dive

The NIOS interrupt receiver is integrated into the NIOS processor and can accept up to 32 interrupt signals. Normally each IP core that uses interrupts will be assigned a single interrupt line. The NIOS II/f processor also supports an external vectored interrupt controller which is significantly more advanced than the internal controller. See [Altera VIC](#).

Before any interrupt code will work, the core you intend to use with interrupts needs to be connected to the NIOS Interrupt Receiver in Qsys as shown below. The number in the connection point defines the interrupt priority and must be unique as shown in Figure 52. It may be counter-intuitive but the **LOWER** IRQ number values have **HIGHER** Interrupt Priority.

	Name	Description	Export	Clock	Base	End	IRQ
	instruction_master	Avalon Memory Mapped Master	Double-click to	[clk]			
	irq	Interrupt Receiver	Double-click to	[clk]	IRQ 0	IRQ 31	
	debug_reset_request	Reset Output	Double-click to	[clk]			
	debug_mem_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0108_0800	0x0108_0fff	
	custom_instruction_m...	Custom Instruction Master	Double-click to				
	jtag_uart_0	JTAG UART					
	clk	Clock Input	Double-click to	altpll_0_c2			
	reset	Reset Input	Double-click to	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0108_1160	0x0108_1167	
	irq	Interrupt Sender	Double-click to	[clk]			
	led_pio	PIO (Parallel I/O)		altpll_0_c2	0x0108_1100	0x0108_110f	
	button_pio	PIO (Parallel I/O)					
	clk	Clock Input	Double-click to	altpll_0_c2			
	reset	Reset Input	Double-click to	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0108_10f0	0x0108_10ff	
	external_connection	Conduit					
	irq	Interrupt Sender	Double-click to	[clk]			
	switch_pio	PIO (Parallel I/O)					
	clk	Clock Input	Double-click to	altpll_0_c2			

Figure 52: Interrupt Connections in Qsys

The interrupt priority is important because interrupts with higher priority take precedence over lower priority interrupts. Therefore only a higher priority interrupt can interrupt a lower priority interrupt. Note that setting interrupt priority in the NIOS processor can only be accomplished in hardware (Qsys).



Watch out!

Putting function calls (such as a printf) within an interrupt service routine is considered very bad form. For example, if you did put a printf inside the ISR C-code for handling the button_pio shown above and the interrupt priorities of the jtag_uart and button_pio were reversed you would have problems. The button_pio would have priority (its IRQ at level 0) and the printf (through the jtag_uart with IRQ at level 1 – LOWER priority than Level 0) would deadlock the system waiting for an interrupt that is lower priority. Always ensure only appropriate code goes into an ISR – no function calls, spin-waits or while statements. ISR rule of thumb – simply get in and get out.

Some cores have the ability to generate interrupts as you have seen during the building of hardware for the lab. Depending on how the core was designed some of the interrupt settings can only be changed in the hardware which is done by accessing the core in Qsys and then regenerating and recompiling the project. Other interrupt settings can only be accessed in software by adjusting registers in the core over the Avalon bus.



Tips and Tricks

For the Altera PIO core, settings such as interrupting on a level or interrupting on an edge can only be changed in hardware. Settings such as which bits in the port will actually generate an interrupt (interrupt mask) can only be changed in software. See [Altera PIO core](#)

Armed with this understanding of hardware side of the interrupts, we'll move on to using interrupts in software.



Take Away

If you want a core to generate interrupts it needs to be connected to the NIOS Interrupt Receiver in Qsys. Interrupt priority is set in Qsys and cannot be changed in software. IP cores can include interrupt settings that can only be configured in hardware or only configured in software or both.

Software Considerations

Addressing interrupts in software is a little more involved than the hardware side. However we can start with the knowledge that our core is connected to the NIOS interrupt receiver and that we have set an appropriate interrupt priority.

There are three main components required in your code to use interrupts:

1. An include for Altera's interrupt controller interface code
2. An interrupt service routine (ISR) which is a special function in your code that is automatically called by the NIOS hardware abstraction layer (HAL) exception handling system when the associated interrupt is triggered and it has the highest interrupt priority.
3. A statement in your main function that registers the ISR with the NIOS HAL exception handling system.

So at the top of your c file you should add the include,

```
#include "sys/alt_irq.h"
```

This "include" gives you access to the required calls needed to manage your interrupts as described below. Somewhere in your code before your main you should put your ISR, a generic ISR would look like this,

```
static void name_of_your_ISR (void* context, alt_u32 id)
{
    //ISR code goes here

    //Command to clear the interrupt goes at the end of the ISR
}
```

Note that the ISR needs to be declared “static” to ensure the compiler does not optimize it away. When you write your own ISR give it a more descriptive name than we have given ours (i.e.: don’t use ‘name_of_your_ISR’). The pointer ‘context’ is provided to pass user specific information to the ISR, it is safe to ignore this as we can get by without it. The ‘id’ is simply the hardware interrupt number of the core that is generating the interrupt, this can also be safely ignored as you won’t need to use it. Write the code that you want to execute during the interrupt inside the ISR. Generally this code is related to data associated with the device generating the interrupt. In the case of a PIO that has buttons connected to it you would probably want to determine which button was pressed in the ISR and respond to or store the data appropriately.



Tips and Tricks

Usually in an ISR you’ll want to signal your main program loop about the data received in the ISR. An often used approach is to declare a global variable in your code to use as a flag that can be evaluated in your main loop. If the interrupt happens the ISR will set the flag and the main program loop will detect it the next time it tests the flag.



Watch out!

If you use a global variable as a flag that is set only in your ISR you should declare the variable as “volatile” to ensure the compiler does not optimize it away.

The most important part of the code in the ISR is the command that clears the interrupt condition and it should be the last command issued in the ISR.



Watch out!

In the hardware you defined in Qsys the interrupts are being generated by IP cores external to the NIOS. Any core that can generate an interrupt for the NIOS must include a way for the NIOS to access it over the Avalon bus and reset the condition that is causing the interrupt. Because every core is different you will need to check the datasheet of the core to determine which register is used to reset the interrupt and add the required code to do so at the end of your ISR. Failure to clear the interrupt in the ISR will result in deadlock; the processor will never return from the ISR.

Below we have created the ISR to handle an interrupt from an Altera PIO core named BUTTON_PIO. For the PIO core it turns out writing anything at all to the base register offset 3 causes the interrupt to be cleared. See [Altera PIO core](#)

```
static void name_of_your_ISR (void* context, alt_u32 id)
{
    //ISR code goes here

    IOWR(BUTTON_PIO_BASE, 3, 0x0);
}
```

Next we need to cover registering the ISR with the HAL exception handling system. The point of registering an ISR is to tell the HAL which ISR to execute for a given hardware interrupt id. Here we show how to register the ISR shown just above:

```
alt_irq_register( BUTTON_PIO_IRQ, (void *)0, name_of_your_ISR );
```

This function is called within main and it registers the ISR by passing in the hardware interrupt number, a null pointer and the ISR name. The null pointer we can ignore because it is simply the pointer 'context' passed into the ISR that we are not using.

Finally we need to know how to enable and disable the interrupts for each interrupt source in the system. Depending on when your "main" c-code allowed to do interrupt servicing of for a core, the "main" c-code will enable or disable the core interrupts as per the individual core datasheets.

If PB0 were connected to the bit 3 input of the Button_PIO, hypothetically speaking, then the following could be used to ENABLE the Button_PIO_IRQ when only PB0 is activated by the code:

```
IOWR(BUTTON_PIO_BASE, 2, 0x4);
```

For further details see [Altera PIO core](#).



Tips and Tricks

You'll want to use the BSP defined hardware interrupt id similar to the registering command above 'BUTTON_PIO_IRQ'. If you forget the BSP define for a core you are registering just search the 'system.h' file in your BSP project in eclipse which includes all the defines for the cores you have added to your hardware in Qsys.

In the case above, if the hardware is properly setup and the interrupt has been registered and enabled then if PBO is pressed the BUTTON_PIO core will generate an interrupt and the code will jump into the related ISR. In a random scenario the ISR code could read the BUTTON_PIO register that is associated with the input data and determine which button was pressed. It could also set a global variable with a number that indicates which button was pressed and then clear the interrupt by writing a value to the third register in the PIO core. At that point the ISR would complete and the code would resume where it left off.



Deep Dive

In actual fact the interrupt handling is a little more complex. When an interrupt is received the program is directed to the exception address where code resides that can handle every type of exception that occurs in a NIOS processor. The handler stores register data onto the stack for safe keeping and then the code starts a process to figure out what just happened. After checking several possible causes it will determine that there is a hardware interrupt event and it will determine which interrupt occurred and run the related ISR. When the ISR completes the exception handler code will restore the register data from the stack and reset the program counter to where it was before the interrupt occurred.



Take Away

There are four key elements required in code to use interrupts:

- 1) An include statement,
- 2) an ISR (which does basic processing and also clears the interrupt at the end of the ISR),
- 3) A call to register the interrupt in the main c-code,
- 4) Enabling/Disabling the interrupt by the "main" c-code inside the peripheral core (see core datasheet) at appropriate times for the application. This usually involves an Interrupt Mask.

Appendix C: Solving NIOS Software Download Issues

The following download process was detailed in step by step fashion in the board diagnostics project so this section will only provide the high level details to aid in troubleshooting download issues. Assuming that the FPGA has been successfully configured with the LogicalStep_top.sof file, successful c-code compiles start by creating a NIOS project and BSP from template as was described in making the board diagnostics project. When you are ready to compile your c-code for the first time three things need to happen:

1. Save the project (this is usually only required for the first build)
2. Generate the BSP
3. Run project as NIOS II Hardware

The third step above actually starts a sequence of events:

1. The BSP is built
2. The project is built
3. A connection to the hardware is made via Quartus
4. The hardware is checked to ensure the hardware on the board exactly matches the hardware described in the BSP
5. The .elf file is downloaded to the soft core processor at the processor is started

All these steps need to happen successfully for your c-code to run. There are reasons that any of these steps could fail and sometimes it is difficult to figure out which step failed or why.

Step 1 Failure: This is an early failure in the build process and Eclipse mentions an error such as,
`make[1]: *** [public.mk] Error 1`

But earlier in the console printout you'll notice it mentions that the BSP needs to be generated. It is often related to recompiling the hardware and forgetting the Generate BSP step before compiling the project.

Step 2 Failure: This failure is usually related to syntax errors in the c-code but Eclipse usually continues through the remaining steps regardless of a failed compile until step 5 when it fails because no .elf file was created in step 2. Use the 'Problems' tab in Eclipse to determine the related problem(s). You can double click syntax related problems and it will show the issue in the code. Syntax errors will be underlined in red in the code and changes to syntax will not be evaluated until the next compile (i.e., red underline will not be removed until after the next

compile). If it seems nothing will eliminate a red underline, even deleting the text completely then be sure to save before recompiling.



Watch out!

Error messages that indicate the .elf file is missing are almost always related to syntax errors in the code and indicate a step 2 failure.

Step 3 Failure: This failure often occurs during the first time downloading software to the board. If the 'Run Configurations' window suddenly pops up during the compile indicating Eclipse can't connect to the hardware follow these steps:

- Press the RESET button on the LogicalStep board.
- In the Run Configurations window click on the 'Target Connection' tab.
- Scroll the window to the right if required and click the 'Refresh Connections' button.
- When done click the 'Refresh Connections' button again if required.
- When the USB Blaster shows up in the lists click the 'Run' button in the bottom of the window

If you get a message that Eclipse can't connect but the 'Run Configurations' window doesn't open then open the window manually from the menu bar, 'Run>Run Configurations' and follow the procedure above.

Step 4 Failure: If you get an error message that indicates a timestamp mismatch the issue is that the software you are trying to download was not compiled for the hardware on the board and there can be many reasons for this.

- You didn't download the hardware to the board before trying to download software
- You downloaded hardware from a different project folder then the software you are trying to download (verify the file path in the Quartus programmer is correct)
- You changed your workspace relative to your project folder
- You copied your project folder moved it to another location
- You renamed your project folder
- You didn't select the correct .sopcinfo file when you created your software project

If you followed the advice to only ever have one project folder and never move it, copy it or rename it then the main cause would be you forgot to download the hardware to the board. If you can't figure out what's wrong and you didn't make the above mentioned mistakes it is recommended that you follow these steps in order,

- Regenerate in Qsys
- Recompile in Quartus

- Generate BSP in Eclipse
- Run as NIOS II Hardware in Eclipse

If this doesn't work you should try and create a new project and BSP from template in Eclipse and be sure that the .sopcinfo file you are using is actually in your project folder – then copy your source files from the old project to the new project.

Step 5 Failure: This step never fails, if you think this step is failing reread the 'Step 2 Failure'.

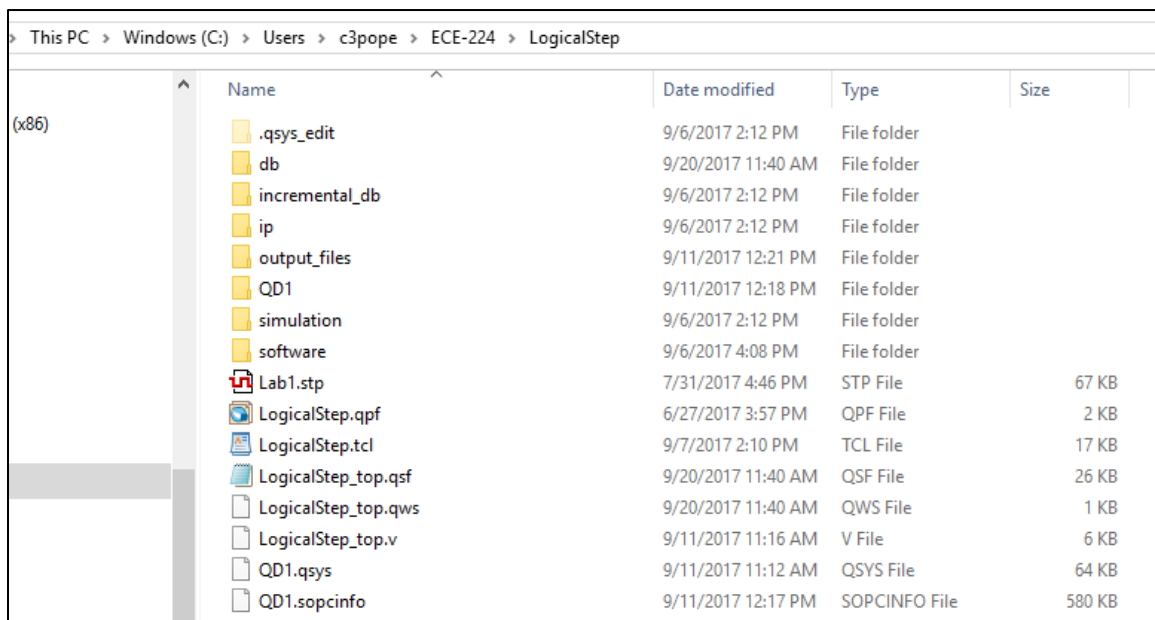
Appendix D: Lab 0 Activity H/W Development Checklist

<input type="checkbox"/> Create an ECE-224 root folder in the location specified by the Lab Instructor
<input type="checkbox"/> Download Lab0 zipped folder file from LEARN into ECE-224. <u>Extract</u> the contents. This will create a "Project folder" for all of the labs called "LogicalStep".
<input type="checkbox"/> Launch Quartus Prime
<input type="checkbox"/> Create a new project by selecting "Files>New Project Wizard". Set the Project Directory to the "Project folder" by browsing to the Project folder and then select that folder.
<input type="checkbox"/> Make sure project name field is "LogicalStep"
<input type="checkbox"/> Make sure top file field is "LogicalStep_top" (please note the lower-case "t" in top.
<input type="checkbox"/> Click "Next" button until "Add Files" dialog appears. Change the file filter to *.*.
<input type="checkbox"/> Add files to the project that you downloaded (LogicalStep_top.v and LogicalStep.tcl)
<input type="checkbox"/> Click "Next" button until "Family and Device Settings" dialog appears.
<input type="checkbox"/> Choose Max10 "10M08SAE144C8G"
<input type="checkbox"/> Click "Finish".
<input type="checkbox"/> Select "Tools>Tcl Scripts.." and then select LogicalStep.tcl, click "Run" (this assigns pins). After this is completed you should see the "MAX 10..." FPGA technology in the upper left of the Project Navigator window.
<input type="checkbox"/> Select "Tools>Qsys"
<input type="checkbox"/> Save Qsys file, call it QD1 (make sure name is correct)
<input type="checkbox"/> In "System Contents" tab, "Description", double click "Clock Source"
<input type="checkbox"/> Make sure Clock frequency is 50000000 (50Mhz)
<i>The following shows the IP Block, Name and Exports for each IP block:</i>
<input type="checkbox"/> Clock Source, clk_50, clk_in="clk_50" clk_in_reset="reset"
<input type="checkbox"/> Avalon ALTPLL, use default name, c0="sdram_clk" c1="audio_mclk" ALTPLL parameters: inclk: 50Mhz, c0 phase shift: -1.5 ns, c1 freq out:12.28 Mhz
<input type="checkbox"/> Uncheck "Create 'locked' output"
<input type="checkbox"/> NIOS II Processor, use default name, no exports
<input type="checkbox"/> SDRAM Controller, sdram_0, wire="sdram_0" SDRAM Controller parameters: bits 16, Chip select 1, Banks 4, Row 12, Column 8
<input type="checkbox"/> System ID Peripheral, use default name, no exports
<input type="checkbox"/> JTAG UART, use default name, no exports
<input type="checkbox"/> Make connections in QSYS as shown in Figure 17 (also check Interrupt priority levels)
<i>Add more cores:</i>
<input type="checkbox"/> PIO (Parallel I/O),, led_pio,external_connection="led_pio" PIO parameters: 8 bits, output
<input type="checkbox"/> PIO (Parallel I/O),, button_pio, external connection="button_pio" PIO parameters: 4 bits, input, Synchronously capture: ANY, Generate IRQ: EDGE
<input type="checkbox"/> PIO (Parallel I/O),, switch_pio, external connection="switch_pio" PIO parameters: 8 bits, input
<input type="checkbox"/> Altera Avalon LCD 16207, lcd_display, external="lcd_display"

<input type="checkbox"/> Audio and Video Config, audio_i2C_config, external_interface="audio_i2c"
<input type="checkbox"/> Audio, Audio (<i>capitalization is important</i>), external_interface="audio_out"
<input type="checkbox"/> UART (RS-232 Serial Port), uart, external_connection="uart"
<input type="checkbox"/> Make connections in QSY as shown in Figure 22 (also check Interrupt priority levels)
<i>Add more cores:</i>
<input type="checkbox"/> Interval Timer, system_timer, no exports
<input type="checkbox"/> Interval Timer, use default name, no exports
<input type="checkbox"/> SPI Master (3 wire serial), spi_master, external="spi_master"
<input type="checkbox"/> Dual 7 Segment, seven_seg_pio, dual_7_segment="segment_drive"
<input type="checkbox"/> EGM, Egm, interface="egm_interface"
<input type="checkbox"/> PIO (Parallel I/O), stimulus_in, external_connection="stimulus_in"
PIO parameters: 1 bit, Synchronously capture: ANY, Generate IRQ: EDGE
<input type="checkbox"/> PIO (Parallel I/O), response_out, external_connection="response_out"
<input type="checkbox"/> Make connections in QSYS as shown in Figure 25 (also check Interrupt priority levels)
<input type="checkbox"/> Auto assign base addresses
<input type="checkbox"/> Set Reset vector in NIOS II processor as shown in Figure 26
<input type="checkbox"/> Set Exception vector in NIOS II processor as shown in Figure 26
<i>Finish rest of hardware project:</i>
<input type="checkbox"/> Click "Generate HDL..." button
<input type="checkbox"/> Save
<input type="checkbox"/> Click "Generate" (if you get errors go back and correct them.)
<input type="checkbox"/> Click "Hierarchy" and change to "Files"
<input type="checkbox"/> Edit the "LogicalStep_top.v" file with the connections to the QD1 instance
<input type="checkbox"/> Copy text from "QD1_inst.v" to the section in the top file below "Place Qsys instance below here"
<input type="checkbox"/> Edit the lines in the LogicalStep_top.v top file: e.g.: .clk_50_clk from (<connected-to-clk_50_clk>) to (clkin_50) etc.
<input type="checkbox"/> Compile the project: "Processing>Start Compilation"
(Ask the TAs or Lab Instructor if you cannot fix any errors generated by this step)
<input type="checkbox"/> Program the FPGA: "Tools>Programmer" (if you do not have a USB Blaster, consult the Lab Manual)
<input type="checkbox"/> When successful, copy your ECE-224 folder tree to your N: drive as a backup.

Appendix E: Lab1 Logic Analyzer (SignalTap)

If you have sufficient hardware resources left in your FPGA (say 25%) you may wish to add an internal Logic Analyzer to your FPGA design for hardware and/or software development and debug efforts. A Logic Analyzer for the Altera FPGA families, called SignalTap, can be used for these purposes. A simple SignalTap design is included on the LEARN webpage for Lab1. To use it you must download the file Lab1.stp from Learn into your FPGA Project Directory “LogicalStep” and then follow some simple steps to include it in your FPGA hardware configuration file. Refer to Figure 53.



Name	Date modified	Type	Size
.qsys_edit	9/6/2017 2:12 PM	File folder	
db	9/20/2017 11:40 AM	File folder	
incremental_db	9/6/2017 2:12 PM	File folder	
ip	9/6/2017 2:12 PM	File folder	
output_files	9/11/2017 12:21 PM	File folder	
QD1	9/11/2017 12:18 PM	File folder	
simulation	9/6/2017 2:12 PM	File folder	
software	9/6/2017 4:08 PM	File folder	
Lab1.stp	7/31/2017 4:46 PM	STP File	67 KB
LogicalStep.qpf	6/27/2017 3:57 PM	QPF File	2 KB
LogicalStep.tcl	9/7/2017 2:10 PM	TCL File	17 KB
LogicalStep_top.qsf	9/20/2017 11:40 AM	QSF File	26 KB
LogicalStep_top.qws	9/20/2017 11:40 AM	QWS File	1 KB
LogicalStep_top.v	9/11/2017 11:16 AM	V File	6 KB
QD1.qsys	9/11/2017 11:12 AM	QSYS File	64 KB
QD1.sopcinfo	9/11/2017 12:17 PM	SOPCINFO File	580 KB

Figure 53: Project Directory with SignalTap File for Lab1

The Lab1.stp file has been setup to monitor 10 signals inside the FPGA for your Lab1 design. With it you can get it to “trigger” on specific events and then have display the internal signal activity just after that event on the SignalTap Console.

For the sake of simplicity and consistency the Lab1.stp file has been designed to monitor just 10 signals inside the FPGA for your Lab1 design. All of these signals are near the top level of the QD1 design (rather than in the depths of the Avalon fabric) to keep things easy to understand. Two of the signals are the “internal” wires for STIMULUS and RESPONSE. The remaining signals in the Lab1.stp have been assigned to the LED’s PIO outputs. Refer to Figure 54.

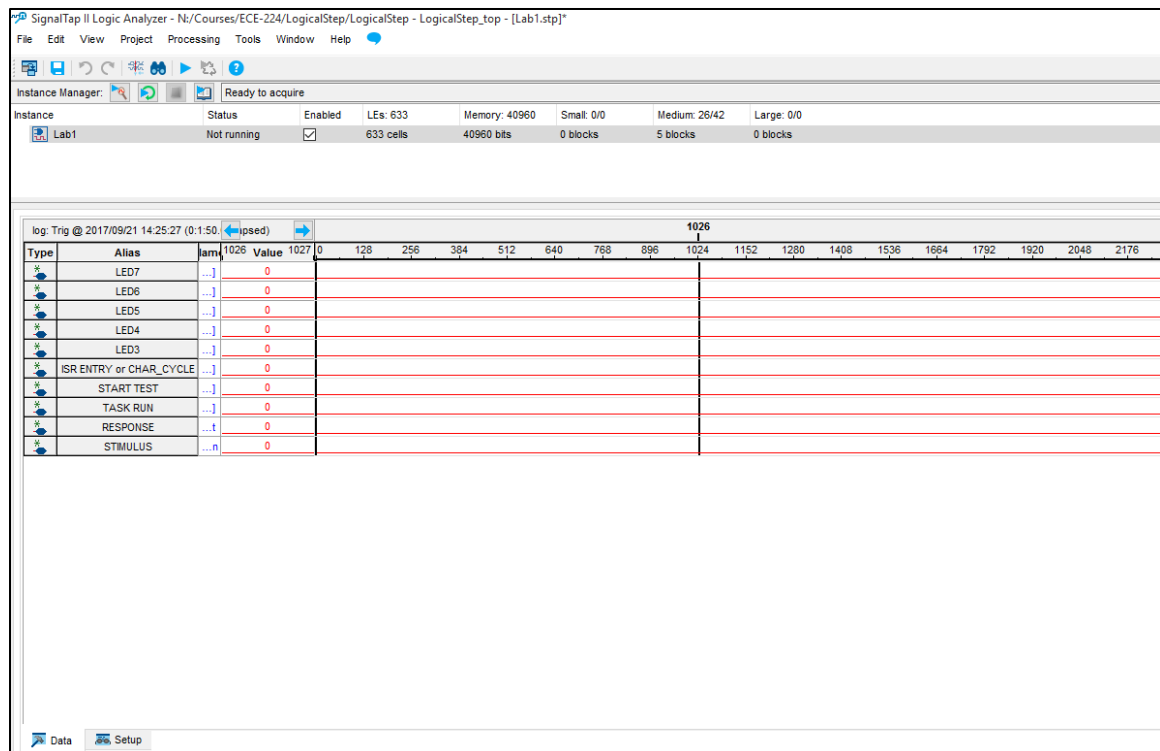


Figure 54: SignalTap Console for Lab1.stp

The SignalTap Logic analyzer can be set to trigger on any of these 10 signals. The LED signals can be used by your NIOS II software to set markers in the execution of the NIOS II program. This is a very useful utility to aid in developing and debugging hardware and software. ALIASES have been entered into the Lab1.stp for easier reference. The LEDS(2..0) connections in Figure 54 have been ‘aliased’ to ISR Entry or CHAR_CYCLE, START TEST, and TASK RUN respectively.

Of course the SignalTap analyzer is far more capable than what is just provided in the Lab1.stp example. If you choose to include it you must do the steps outlined below and then do an FPGA recompile. Also because of a recompile you will have to regenerate any BSP that in the Altera/Eclipse NIOS II SBT that will use the new FPGA.

Step1: Inside the Project Directory open Quartus and your LogicalStep Project by just “double-clicking” on the LogicalStep.qpf file. This will launch Quartus and will set your project reference point to the LogicalStep directory and it will load your LogicalStep project. Then within Quartus with go to the “>Project>Add/Remove Files in Project” Tab. A “SETTINGS” dialog window will appear. Then select the “Files” entry in the Category pane of the SETTINGS window. For the “File name:” field press the Browser button to its right and then browse to the Lab1.stp file

(make sure that the Files Explorer filter is set to “All Files”) in your Project Directory and select OPEN. Then you must press the “Add” button in the Settings Dialog window. Refer to Figure 55.

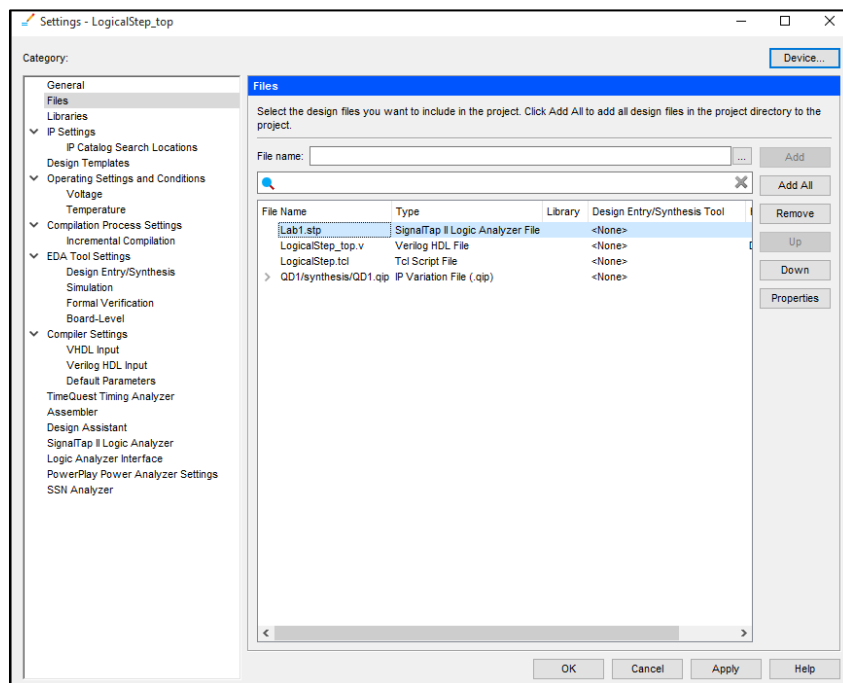


Figure 55: Inclusion of the Lab1.stp file in your LogicalStep Project

Step2: To enable the operation of the Lab1.stp you will then browse in the Category Pane of the same Settings Dialog window to the “SignalTap II Logic Analyzer” option and then select it. A new Dialog window becomes visible. Please refer to Figure 56. Click on the box to Enable SignalTap II Logic Analyzer and the Lab1.stp file should be seen in the SignalTap II File Name field. Click OK.

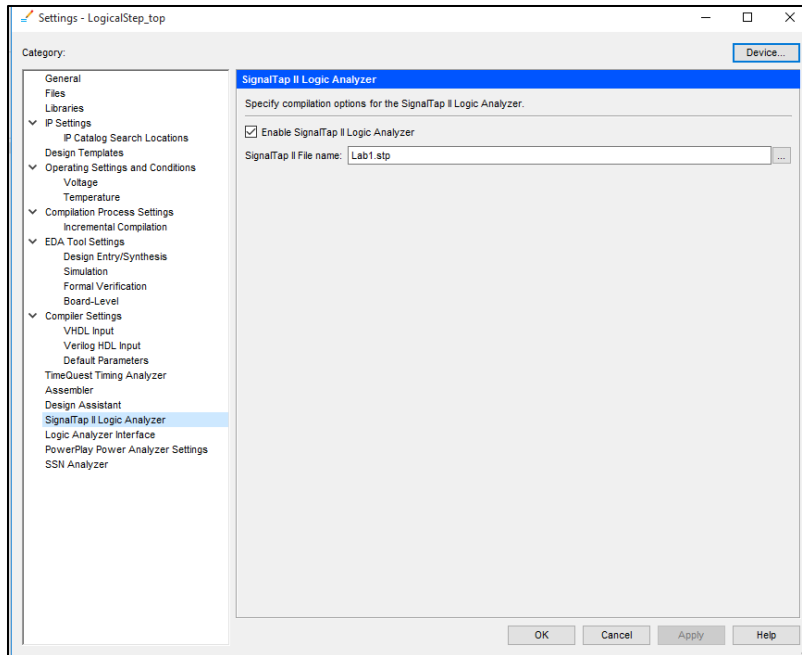


Figure 56: Enabling the SignalTap Logic Analyzer for the FPGA

At this point the SignalTap analyzer is now included in the FPGA design. Perform an FPGA recompile. Then you may now download the just-compiled FPGA file to the LogicalStep board and then close out the programmer.

After it is downloaded you may invoke the SignalTap Logic Analyzer by selecting >Tools>SignalTap II Logic Analyzer. The Lab1.stp version of the console should appear like in Figure 54.

There are numerous aspects to the operation of the SignalTap Analyzer. A great number of them have been “FIXED” by the Lab instructor for ease of use. An appnote is included in the Lab Study 1 area on LEARN for the course. The setup for choosing the signal and active edge as a “TRIGGER” can be followed from that document. The Lab Instructor can provide further information on the use of this powerful utility.

As mentioned earlier after the compile of the FPGA is completed you will have to have to regenerate your BSP files in the Altera/Eclipse Nios II SBT utility that uses the updated FPGA. After the BSP regeneration(s) are done you can add or modify your C-code to set markers for your software when it is running. These are very useful when you see how the markers are

displayed in time against the other aspects of the FPGA operation (the asynchronous Stimulus for example).

Generally speaking, to use a marker you can set a LED bit to ON at the beginning of an ISR or background task and then turn that same LED bit to OFF when it is completed. Thus you can track in time when those software activities are running in comparison to the other monitored signals connected to the SignalTap analyzer.

An example of a captured event during Interrupt Sync Mode is shown in Figure 57.

For this SignalTap capture the C-code was set up to set LED2 to ON and OFF when the Interrupt Service Routine (ISR) was started and ended. LED1 was used to signal the Start of a Test (Turned ON and then OFF just at the beginning of the test). LED0 was used to indicate the beginning and end of a background task. The Stimulus and Response signals are monitored from those respective PIO input or output respectively.

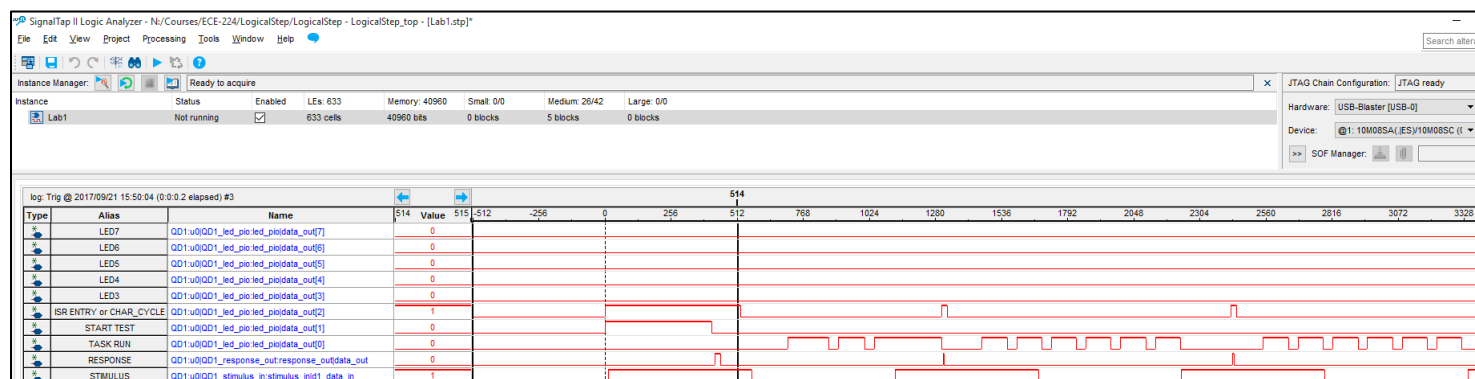


Figure 57: SignalTap Capture

Appendix F: Lab2 Hardware Testing

Notes on Testing your LogicalStep Audio Hardware

(Extract from developments notes of a previous project – Jim Miller)

The audio pipeline on the LogicalStep board can be tested by connecting the audio input of the board to the audio output of the computer and the audio output of the LogicalStep board to speakers or headphones. The IP core provided by Altera has FIFO's in which the audio data is stored to be spooled out at the appropriate sampling rate to the audio codec on the board. The FIFO space must be carefully monitored to ensure that it does not overflow or underflow with audio data. Later, when implementing the full audio player, errors here will lead to distortion of the audio.

- Within the Altera/Eclipse utility NIOS II SBT create a new software project. Using the “Hello World” template, give your first Lab2 project a name such as “Lab2_Loop_Audio”. Step by step instructions to do this were given in Lab 1.
- Inside the Project pane of Altera/Eclipse NIOS II SBT you will see the Lab2_Loop_Audio software project folder. Expand that project and open the “hello_world”.c file. Replace its contents with the contents of the c-code example given in Figure 2 at the end of the Altera IP Core datasheet document. Don't forget the new #include file. You can do this by copying the sample code on the last page of the Altera Audio Core data sheet (see Figure 2) into a text editor such as Wordpad initially. This converts the pdf file contents to printable characters. Then copy the same code from that Wordpad editor and insert it inside the hello_world.c by overwriting the original c-code that is in there. You can then discard the Wordpad file.
- Inside the NIOS SBT compile your Lab2 software project and run it on the board. Connect headphones (or speakers) to the “Line Out” audio jack and a source (the computer, your phone etc.) to the “Line In” jack. When you play the audio on the source you should now be able to hear it through your headphones. If you cannot hear the audio, there may be a problem with your test code or your hardware from Lab0 or Lab1. You are encouraged to seek assistance from the teaching team to resolve your issue.



Take Away

Before you begin work on your audio player ensure your audio pipeline is functional.

Appendix G: Lab2 FatFS Commands

FatFS Command Line Interface – Tips and Tricks - By Jim Miller

When the project starts it will print out the help screen – you can reprint the help screen by typing “h” and then enter. If you make an error typing the commands simply hit enter and retype the command. Don’t try to correct it using backspace as this will not work.

The main CLI commands you need to execute to access the music files on the disk are:

1) To initialize:

“di 0” – this is ‘disk initialize’ and mounts the drive to the ‘0’ location.

“fi 0” – this is ‘force initialization’ of drive ‘0’ location.

“fl” – this is ‘file list’ and will print the file names and other data that are on the SD card

2) To select an audio file for playback:

“fo 1 dead_jim.wav” – this is ‘file open’ with ‘1’ read privilege and opens ‘dead_jim.wav’.

3) To play the selected audio file:

“fp 626798” – this command plays the opened file by reading 626798 bytes (which is the file size of ‘dead_jim.wav’) from the SD card while spooling it to the audio codec – **note that this function needs to be written by you.**

Below is a screen capture of the NIOS Console that shows the complete process from mounting the drive to playing an audio file. The ‘rc=0’ lines etc. below are responses from the CLI program.

```
FatFs module test monitor
LFN Disabled, Code page: 1250
dd <phy_drv#> [<sector>] - Dump sector
di <phy_drv#> - Initialize disk
ds <phy_drv#> - Show disk status
bd <addr> - Dump R/W buffer
be <addr> [<data>] ... - Edit R/W buffer
br <phy_drv#> <sector> [<n>] - Read disk into R/W buffer

bf <n> - Fill working buffer
fi <log_drv#> - Force initialize the logical drive
fs [<path>] - Show logical drive status
fl [<path>] - Directory listing
fo <mode> <file> - Open a file
fc - Close a file
fe - Seek file pointer
fr <len> - Read file
fd <len> - Read and dump file from current fp
fz [<len>] - Get/Set transfer unit for fr command
h view help (this)
```

di 0
rc=0

fi 0
rc=0 FR_OK

fl
----A 2012/10/11 16:38 626798 DEAD_JIM.WAV
----A 2012/10/11 16:38 1640572 DOCTOR.WAV
----A 2012/10/11 16:38 755822 FULL_POW.WAV
----A 2012/10/11 16:38 1548398 GOOD_BAD.WAV
----A 2014/03/18 12:01 5301020 GSHELTER.WAV
----A 2012/10/11 16:38 553070 KIRK2ENT.WAV
----A 2014/03/18 12:01 74926244 LONG.WAV
----A 2010/07/15 17:18 2420164 LRSTER.WAV
----A 2012/10/11 16:38 4008402 MED.WAV
----A 2005/11/08 19:59 1040836 MUSIC.WAV
----A 2008/03/28 14:44 3611756 STEROTST.WAV
----A 2016/09/04 13:18 3710268 R_ROLL.WAV
----A 2012/10/11 16:38 1400942 THANKU.WAV
----A 2014/01/08 16:53 1541764 V3FPS.MPG
----A 2014/01/08 16:53 7842516 VLONG.MPG
----A 2014/01/08 16:53 2166960 VSHORT.MPG
----A 2012/10/11 16:38 218066 WINDOW.WAV
17 File(s), 113313598 bytes total
0 Dir(s), 1893990400 bytes free

fo 1 dead_jim.wav
rc=0 FR_OK

fp 626798

fo 1 thanku.wav
rc=0 FR_OK

fp 1400942

Appendix H: Lab2 Logic Analyzer (SignalTap)

The SignalTap Logic Analyzer that is provided for Lab2 is available on LEARN inside the Lab2 Study page. You can download it into your LogicalStep project directory and set it up similarly to what was explained in APPENDIX E. This SignalTap utility version can be used to see activity on the Audio/Codec, LCD and Push button interfaces.

There are three instances of Logic Analyzers in the Lab2.stp file. However only ONE can be used at a time for triggering and displaying signal activity. To choose the desired one just “double-click” on the instance. The one for the Push Buttons is shown in Figure 58 below.

USER LED’s are provided for you in each instance to use as software markers as was done for Lab1. You might activate one or more of these LED’s to indicate ISR activity etc.

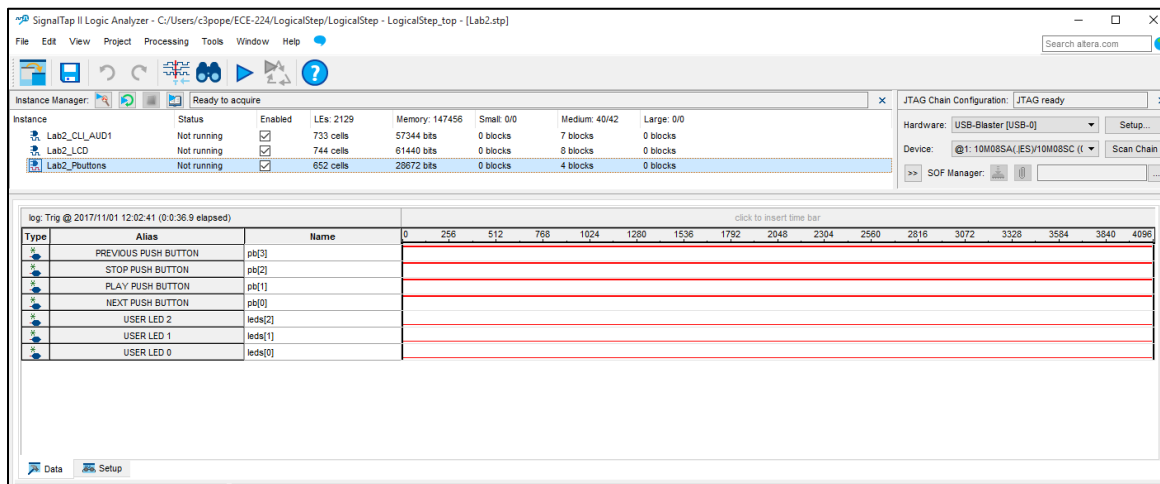


Figure 58: SignalTap Console for Lab2.stp