

# MicroBlaze EDK tutorials

The MicroBlaze EDK tutorial is motivated by the fact that the EDK environment is complex to start with. Furthermore, the GECKO3/4 platform is HuCE-microLab's education and engineering platform therefore new members and students need a straightforward tutorial to familiarize with the topic.

- [MicroBlaze EDK tutorial](#)

## Tutorial topics

- [Tutorial introduction](#)
  - Versions
  - Platform modules
  - Exercise information
  - How to use this tutorial
- [Tutorial chapter 1](#)
  - Open a terminal on a Linux OS
  - Create an XPS project using the Base System Builder (BSB)
  - Create a simple hardware design using Xilinx and GECKO3 IPs available in the Embedded Development Kit (EDK)
- [Tutorial chapter 2](#)
  - Adding IPs to a hardware design
  - Implement the design by using ISE
- [Tutorial chapter 3](#)
  - Add the custom peripheral to your design
  - Write a software application
  - Generate the bit-stream and verify the operation in hardware
- [Tutorial chapter 4](#)
  - Add an internal RAM memory controller block
  - Write an application to access an IP peripheral
  - Modify a linker script
  - Partition the executable bitfile in LMB and PLB memory spaces
  - Generate the bitstream and verify the operation in hardware
- [Tutorial chapter 5](#)
  - Using an XPS timer
  - Assign an interrupt handler to the XPS timer
  - Create an Interrupt handler routine
  - Use the SDK debugger
- [Tutorial chapter 6](#)
  - Add ChipScope Analysers into a system
  - Cross debug with ChipScope Analyser and the SDK debugger

# Introduction

The tutorial, which contains several exercises, is based on the **Xilinx Spartan-3E Starter Kit Tutorial** and is adapted by the author to the special needs for the **GECKO** project as well as for teaching purposes. To get more information about Xilinx educations material, you are requested to have a look on [Xilinx tutorial](#).

## Version verifications

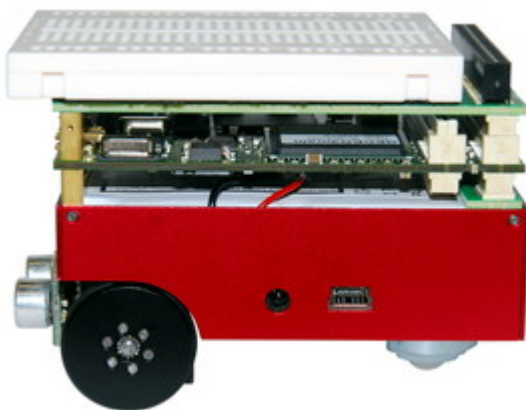
This tutorial has been verified on the following hardware, software, firmware and IP versions:

Software	
tool	version
OS	Ubuntu 12.04.1 LTS (tested with Gnome and KDE 4.1)
Kernel	2.6.32-25-generic
Xilinx ISE	11.4
minicom	9.1.03i Build ISE J.33
cuteCom	2.2 (compiled Mar 7 2007)
Hardware	
componen	version
Gecko3EDU	revision: 1.0

## Platform modules

To complete the following labs some GECKO3 components are required. First, you work with the GECKO3main board. This is a powerful FPGA based development board containing a couple of components such as memory elements (Flash or RAM), interface elements (GPIO, RS232 or Ethernet) and a partially general purpose bus (GPBUS).

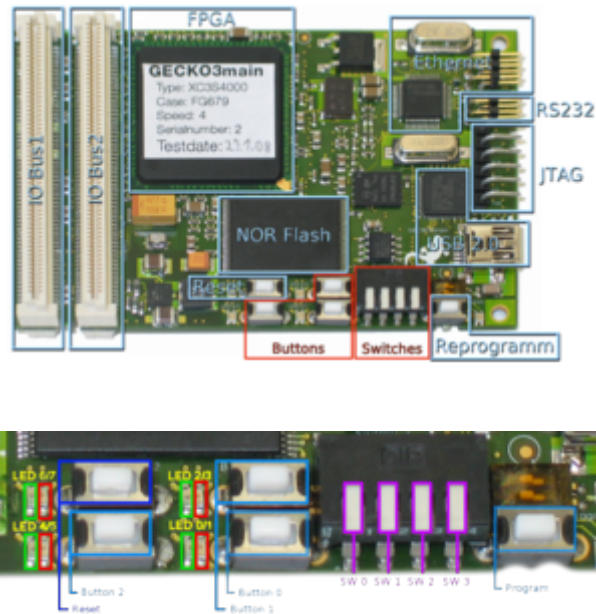
For the powering, the GECKO3main requires an Add-On board like the GECKO3Robot or GECKOXSimple. Finally, to get an impression about the purpose of this components, the development team built up the GECKO3EDU that is thought for motion exercises, mainly, as shown in the figure below.



Depending on the set-up (GECKO3Robot or GECKO3Simple some UCF ports e.g. RS232 must be adapted to match the bus specific adaptations.

## Connections to GECKO3main

In the figure shown below are the most important components highlighted. Be careful when connecting to JTAG. If you use the USBPlatform Programmer instead of the parallel port cable, check the connector as well as the cable labels.

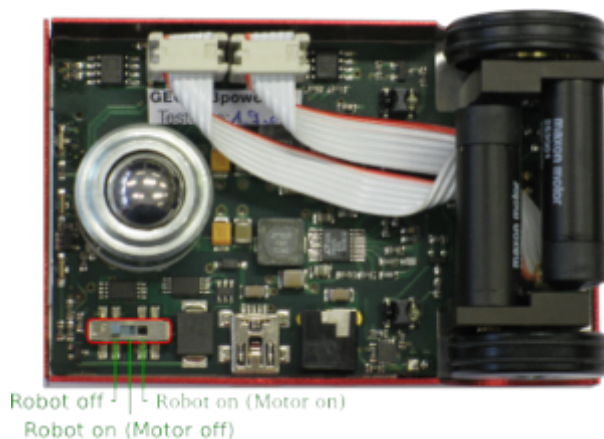


Labels must match!! It will not work, otherwise.





Be aware of the different UCF settings depending on the setup (Xsimple or Robot as bottom Add-On).



## Naming conventions and marks

There are some conventions especially on naming that are intended to be consistent throughout the tutorial. Manufacturer and product names are formatted in accordance with the standard rules of English grammar, e.g. **“This is an example”**. Manufacturer and model names are proper nouns, and thus are written bold and beginning with a capital letter, e.g. **MicroBlaze**. The tutorial has some sections, which are highlighted by notification boxes. Do not skip those boxes. The additionally given notes are important and had been highlighted, therefore. The following boxes are used:

**Notes:** This indicates a note. Notes are used to mark information that could help you or indicate possible “weirdness” in a specific exercise as well as in a sub-part of an exercise. Therefore, a note provides additional information or helpful links.

**Warning:** This is a warning. In contrast to notes mentioned above, a warning should be taken more seriously. While ignoring notes will not cause any problems conversely ignoring warnings could cause serious problems.

□ Exercise: This icon marks a section that is intended especially for students. The exercises are for checking the newly gained knowledge during the previous steps. It is important to do this parts patiently for getting a solid and well formed basic knowledge about the GECKO3 and the work with FPGA development as well.

## Special environment settings

When working remotely (on one of our servers over a SSH channel) some special settings are necessary. Next, find described the most known issues and information about a possible solution.

### Problems with the programmer

Usually, the target is connected to the computer where Xilinx tool-chain is installed. In this environment the pre- and post-processing will be done on the server. Hence, the target is not connected with the server physically. Therefore, the download process must be performed manually by using iMPACT. For more information read the appropriate section in the first laboratory exercise.

### HTML documentation problems

The API documentation is available from the server, where Xilinx tool-chain is installed. For using the HTML documents the browser e.g. FireFox has to be started with the **"-no-remote"** option. Add the option under *Edit* → *Preferences* → *HTML-Browser* . A correct HTML-browser setup line looks like the following:

```
/usr/bin/firefox -no-remote
```

Delete unnecessary options and settings.

# Lab1: Simple Hardware Design

## Introduction

The first lab guides you through the process of using Xilinx Platform Studio (XPS) to create a simple soft core based processor system targeting the GECKO3. The GECKO3 platform is a HuCE-microLab project published under the CC. For further information do not hesitate to visit the project web page [GECKO3 wiki](#) or the public repository on [OpenCores](#).

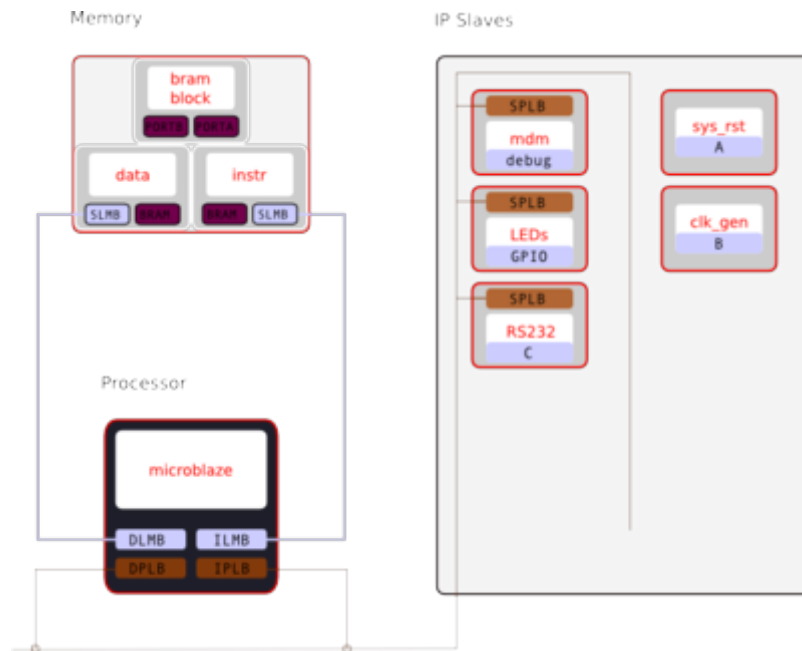
## Objectives

After working out the first lab, you will be able to:

- Open a terminal on a Linux OS (Debian based)
- Create a XPS project using the Base System Builder (BSB)
- Create a simple hardware design using Xilinx and GECKO3 IPs available in the Embedded Development Kit (EDK)

## Procedure

The purpose of this chapter is to work through a complete hardware and software processor system design flow. For better understanding, each laboratory exercise is based on the previous. The figure below represents a design overview, which you are going to build by completing all required steps of laboratory one. The components, marked by a red border, will be added while completing the tasks described. Others have been added during previous steps or by a template, respectively. The block-structure shown in the figure below is one way of describing a SoC systems. If you work on a project, it is important to describe a system by blocks in order to have a structured design. A straight forward design practice is the **top-down** and **bottom-up** methodology. The SoC platform XPS provides by its IP library a method of building an embedded system in the bottom-up methodology that suppose a well developed IP library. Each IP core has its data sheet where border conditions and specifications are summarized.



In this exercise, you will use the BSB to create a soft-core processor based system consisting of the following processor IP:

- MicroBlaze (soft-core processor)
- PLB (Processor Local Bus)
- UART for serial communication (Universal Asynchronous Receiver Transmitter)
- BRAM (block RAM)
- GPIO for LEDs

First-of-all, you will learn how to open a Linux terminal (also called shell). Secondly, you will create a project using the Base System Builder (BSB). Furthermore, you analyze the created project with some guidelines. Finally, the first project will be generated based on the introduced basic elements.

## How to complete the labs by using Linux

### How to boot Linux at BFH

As you may know, you work with a Ubuntu Linux installation personalised for HuCE-microLab needs. Take this into consideration when completing the exercises during spear-time with your own Linux installation. For convenience, however, it is required working with the BFH installation, which is carefully supported thought the HuCE-microLab staff in cooperation with the BFH ITS Linux team. The BFH installation is based on a PXE set-up that means the system will boot fully over the network. However, any changes to the system will be deleted after a reboot, so store every bit on your personal home drive. Normally, the computer's settings are set for a PXE boot, if this is not the case inform the course responsible.

### Open a terminal by using gnome

Open a terminal by selecting **Applications → Accessories → Terminal**. For additional information have a look in figure shown below. (If you prefer having a starter on the panel, feel free to create one. The short-cut in Ubuntu is [Ctrl]+[Alt]+T)



The figure below shows a terminal of an Ubuntu Linux with the Gnome window management system. Next, you should connect to HuCE-microLab's server, where the IDE runs much faster than on a standard PC. To do this, establish a ssh connection to the server - name given by the teacher. At the verification date the server name was xena but that changes from time to time. Anyway, use first the command `ssh -X xena.bfh.ch` for more information to the command ssh use the manual page.

```
man ssh
```

```
ssh -X xena.bfh.ch
```



Now, start the Xilinx Platform Studio (XPS) by the command **xps &**.

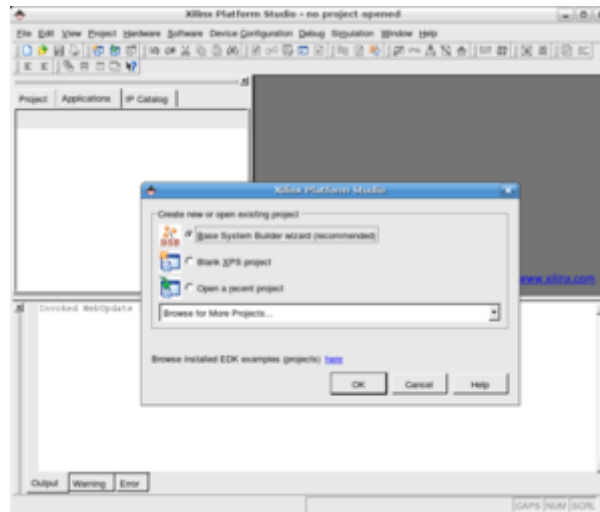
```
xps &
```

The command **xps** launches the program and the symbol **&** puts the process - sequence of interdependent and linked procedures that, at every stage, consume one or more resources - in the background. If the program runs in background mode the terminal can be used for other system interactions. If you close the terminal every program started from this terminal will be terminated instantly.

## Create a project using the BSB wizard

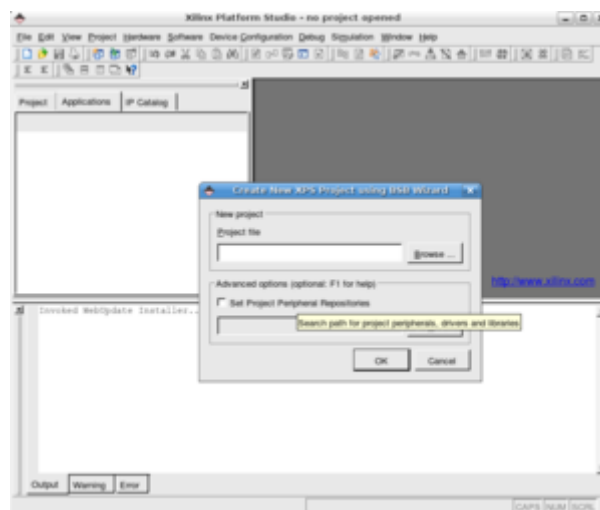
After starting XPS you will see the BSB wizard appearing as a pop-up. Click on the OK button to create a project using the Base System Builder.





Using the BSB is a comfortable way building up a system in no-time. The BSB provides a interface where the hardware can be described by a settings file. This syntax is not part of the tutorial but take it into consideration when working with custom boards.

A lot of Xilinx supported boards are shipped by the software and therefore integrated in the IDE. For custom hardware, the developer has to spend some effort on describing the board in the appropriate way. However, the necessary file is there and ready to use. Get more information by the BSB tutorial. To start the BSB dialogue manually select **File → New Project** or by the shortcut [Ctrl]+[Shift]+[N].

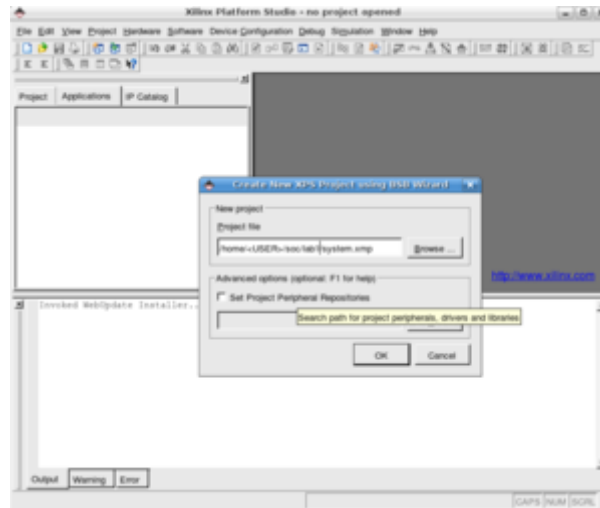


Browse to your home directory and create a folder for the System-on-Chip course and for each lab exercise as well.

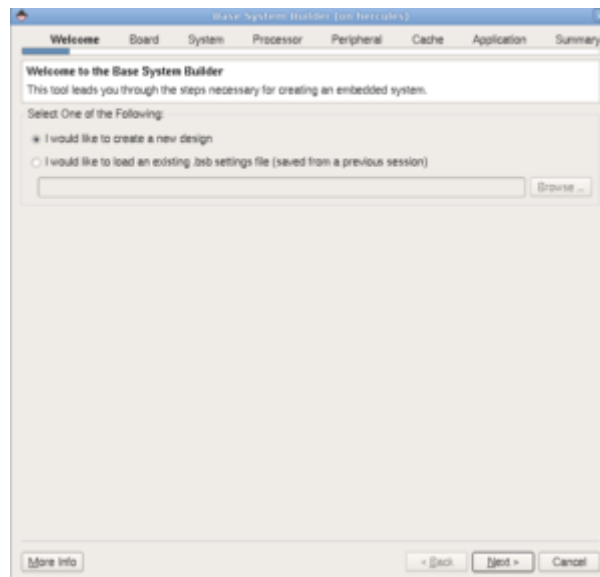
It is best practice creating for each lab a new and unique sub-directory.

The resulting work path is named as following or should have at least a similarity to the suggestion given, next.

```
/home/${USER}/SoC/lab1/system.xmp
```



Click on the [OK] button to display the **Welcome to Base System Builder** dialogue box and select then: “I would like to create a new design”, afterwards click the NEXT button as shown in the figure below.

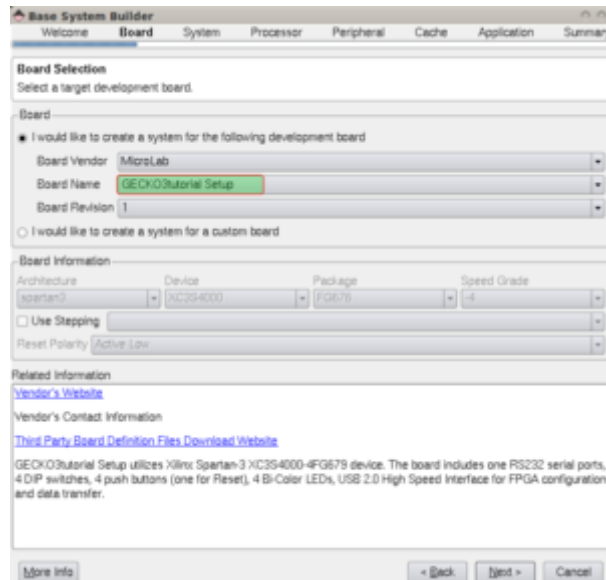


On the following dialogue box you can chose your target architecture. Specify the settings to match the following:

- Board Vendor: MicroLab

For the SoC course use the BSB **GECKO3tutorial Setup**

- Board Name: **GECKO3main Board** or **GECKO3tutorial Setup**
- Board Revision (Verify on board): 1



To get an overview of the board, read the board description.

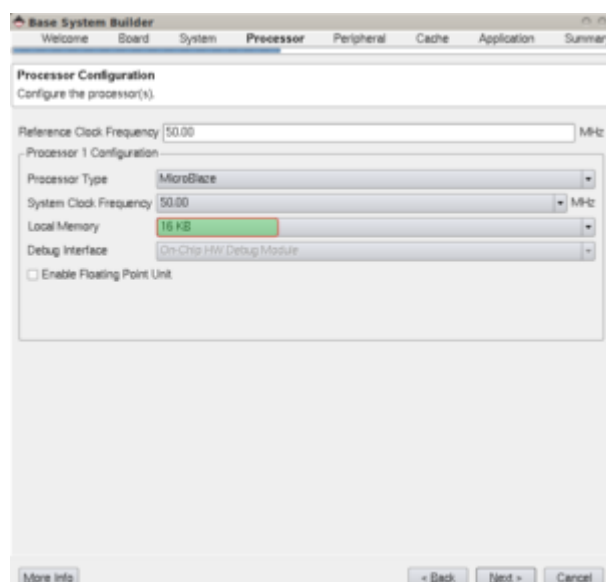
On the next dialogue box, select a single processor design.

In the box named “Configure MicroBlaze”, verify the settings to match the following:

- Reference Clock Frequency: 50 MHz

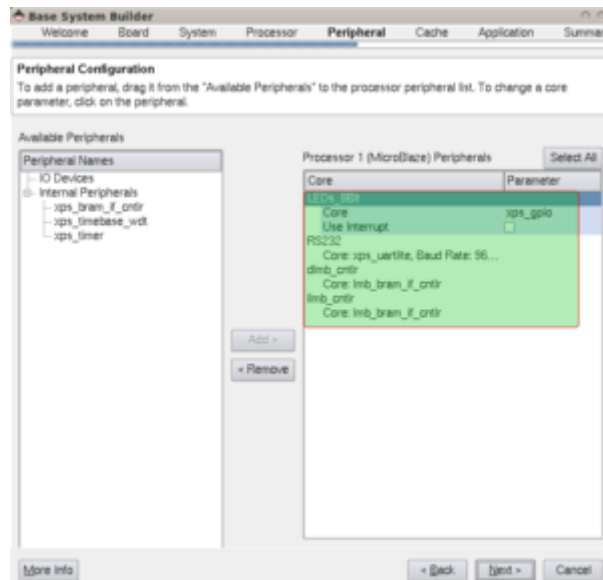
This is the external clock source on the board you are using. This clock will be used to generate the processor and bus clocking. The values settable there may depend on the board you are using due to certain on-chip resources (DCMs) that are used for clock division or multiplication, respectively.

- Processor Type : MicroBlaze
- System Clock Frequency: 50 MHz
- Local Data and Instruction Memory: 16 KB
- Debug IF (interface): On-Chip H/W debug module
- Enable floating point unit (FPU): unchecked

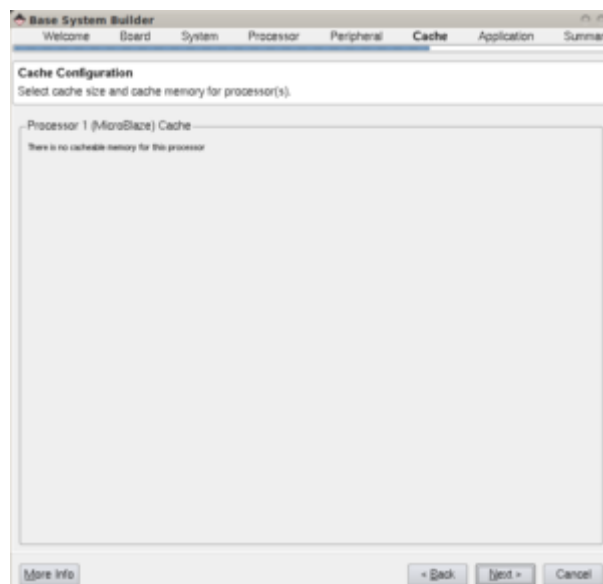


Make sure the **FPU** is unchecked to speed up the build process - keep out unnecessary features. If you need more information or if you feel like getting more details about what you are doing click the button named “More Info”. However, the tutorial was build-up to get a brief introduction with sufficient information to complete each step without reading the additional information notes. Click **NEXT** to display the “Peripheral Settings” dialogue box. Make sure that every removable component is removed otherwise remove them except those listed below. You will end up with a set-up similar to the one shown in the figure below.

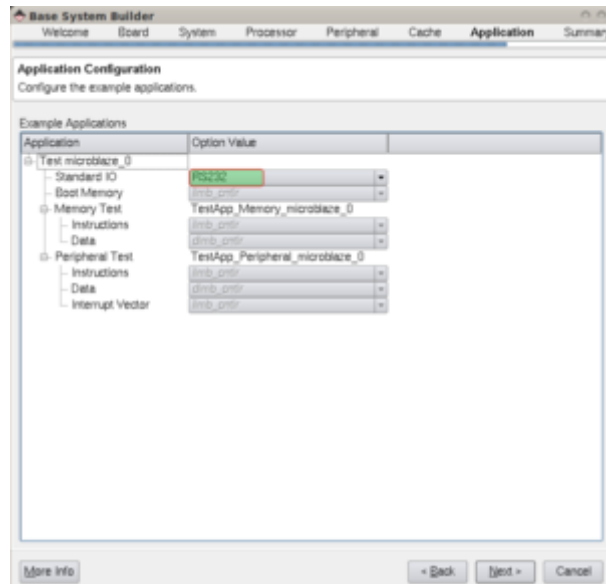
- LEDs\_8Bit
- RS232



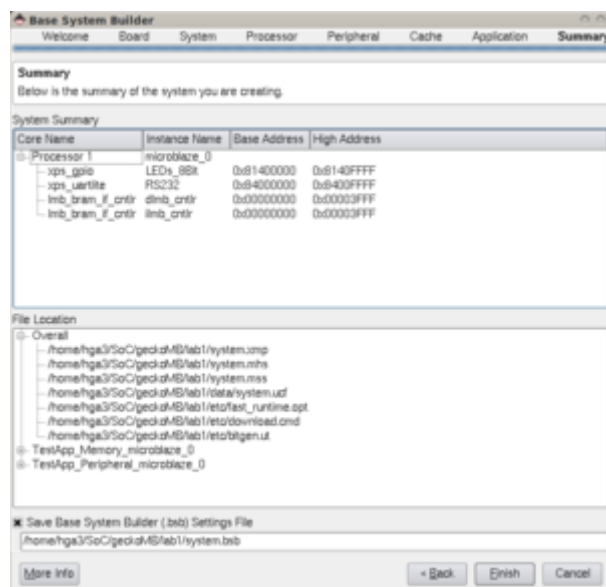
Click **NEXT** to display the Cache dialogue box. For information only, this configuration needs no selection or activation for cache elements.



Click **NEXT** to display the “Application” dialogue box. Make sure the standard IO is set to RS232 due to the fact that the communication will established over the RS232 interface, later. (The mdm as standard IO is not going to be used during the first few exercises.)



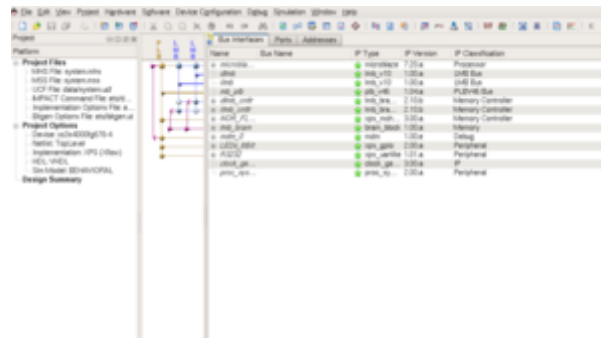
Click **NEXT** to display the Summary dialogue box that summarizes the system being created.



Click on **Finish** to complete the build process supported by the XPS wizard. In case you started XPS for the first time the dialogue shown below appears. Tick the appropriate option in case you wish XPS remember your selections.



The “System Assembly View” will be displayed that shows peripherals and buses used by the system with their connectivity to each other.



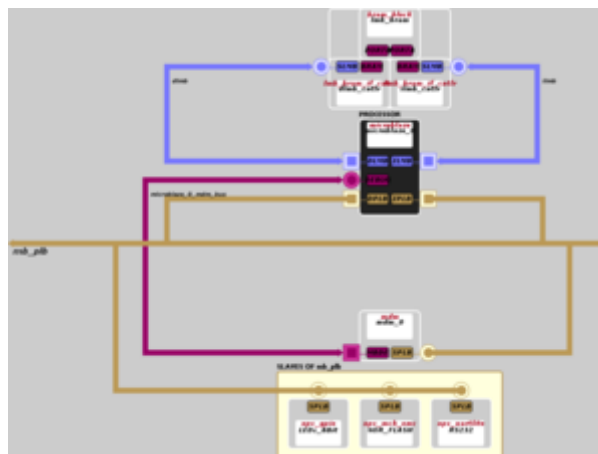
Before building the embedded system, some changes are required in case the GeckoXsimple board is used as the bottom board for powering as well as communication. The changes must be made in the user constrain file (UCF). This file is located in the folder "<workspace>/data/system.ucf". Copy the two lines defining the RS232 communication lines RX and TX. Do not delete settings, comment out changes to the originals. A change to the FPGA pin location used for RX and TX is needed, only. Use for TX LOC = K25 and for RX LOC = K23. The changes are necessary in order to a slightly different communication interface set-up used by the GeckoXsimple.

```
Net fpga_0_RS232_RX_pin LOC = K23 ;
Net fpga_0_RS232_TX_pin LOC = K25 ;
```

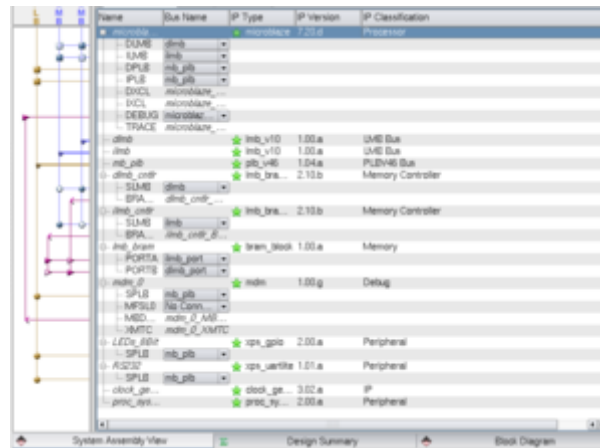
## Analyse

### An overview of important boxes and information windows

Click **Project → Generate Block Diagram Image** as for generating the actual block diagram. To view the new generated diagram click on the "Block Diagram" tab as shown in the assembly view figure. Check the various components used in the design and study the legend as well.

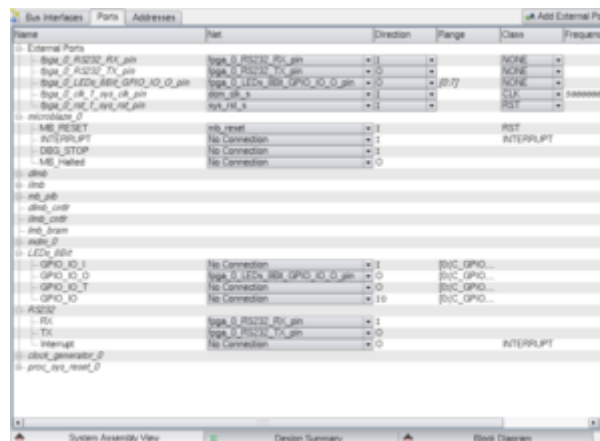


Use the scroll bars to navigate in the block diagram. You will see the soft-core processor **MicroBlaze**, the **LMB** controller and the **PLB** bus connected to the **MicroBlaze**. Just in brackets, the module **mdm\_X** is a abbreviation for MicroBlaze Debug Module. The isolated elements, the ones without any bus connection, are IPs that are not connected to any peripheral bus e.g. DCMs (Digital Clock Managers).



In the “System Assembly View” click on the plus button in front of the name to expand the element of interest. Study the detailed bus connections.

Click on the ports filter to get a overview to the port connection as shown in the figure below.



During the analyze exercises you will learn how to use the filters in the System Assembly View. Beside the three main filters (Bus Interface, Ports, Addresses) are detail filters available. You find them in the right window called "Bus Interface Filters". Play a little bit with those filters to get a feeling for them. Filters make the work easier and the overview about your embedded design much better so learn how to use them.

## Analyse exercise

**List the bus connection to the following peripherals:**

- ☐ debug\_module :
- ☐ dlmb\_cntlr :
- ☐ RS232 :

**List the nets which are connected to the following ports:**


- ☐ RS232 SPLB\_Clk :
- ☐ RS232 RX :
- ☐ RS232 TX :
- ☐ LEDs 8Bit GPIO :

**Select Addresses filter and list the address for the following instances:**

- ☐ RS232 Base address :
- ☐ RS232 High address :
- ☐ dlmb\_cntlr Base address :
- ☐ dlmb\_cntlr High address :
- ☐ ilmb\_cntlr Base address :
- ☐ ilmb\_cntlr High address :

## Generate the net-list

To generate the net-list of the project, follow the next few steps accurately. After every step the message "Done!" will be printed on the EDK terminal otherwise the output "Error!" followed by some information describing the error more detailed. As for checking the build process have a look at the blue circle in the bottom right corner.

1. In XPS, select **Hardware → Generate Net-list** or click on the icon  on the tool-bar
2. Observe the net-list generation in EDK's integrated console window
3. Open a file browser
4. Browse to the **Lab1** project directory
5. Several directories containing VHDL wrappers and implementation net-lists have been created

### List the directories that were created:

- ☐ Directory list :

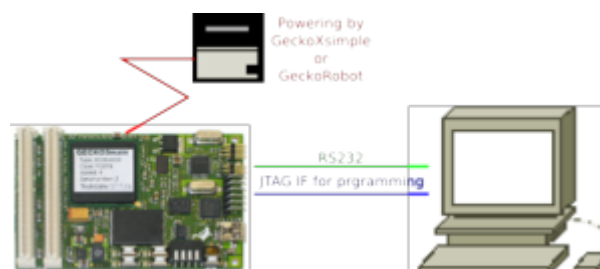
## Generate the bitstream and download the application

1. Connect and power up the GECKO3EDU / GECKOXSImple
2. Open a serial-terminal session and make the right configurations as described below.

For the configurations have a look in the IP core settings of the RS232 IP.

1. Select **Device Configuration → Update Bitstream**

Any problems with downloading. No-worries, check the connections to match the one given by the [intro](#) chapter. Additionally make sure iMPACT is started locally and not on the remotely. Best way is using the starter in the HuCE-microLab application menu.

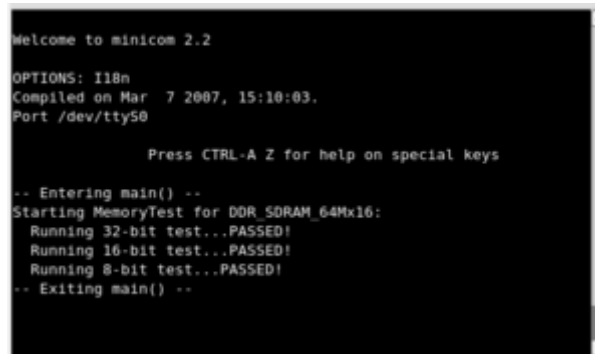


If every thing works fine, you should see the following output on your preferred serial-terminal program. The



output depends on the configuration hence the result may be different. Definitely the lines **“Entering main()”** and **“Exiting main()”** are plotted and if you see them it is fine.

Some hints in case of an error. Check the RS232 connection first. Furthermore, check if the serial-terminal program is listening on the right device node. Additionally, check if the settings for the serial communication are right, have a look in the description following next.



```
Welcome to minicom 2.2

OPTIONS: I18n
Compiled on Mar 7 2007, 15:10:03.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys

-- Entering main() --
Starting MemoryTest for DDR SDRAM_64Mx16:
Running 32-bit test...PASSED!
Running 16-bit test...PASSED!
Running 8-bit test...PASSED!
-- Exiting main() --
```

## Use a serialterminal

To transfer data to and from GECKO3 over the serial interface, you need a terminal program as minicom or cuteCom. Read the following descriptions carefully about the Linux serial interface. Afterwards you will have the knowledge for working with one of the explained serial-terminal tools. Minicom is explained here in order you will maybe work in future in the area of embedded engineering. However, by working as embedded software engineer it could happen that you work on terminals without a running X-Server as for resource saving purposes. In such a situation “minicom” is your choice.

## Information about the Linux serial I/O

An I/O port is a way to get data in and out of a computer. There are many types of I/O ports such as serial ports, parallel ports, disk drive controllers, Ethernet interface, etc. We are going to work with serial ports since modems and terminals are serial devices. Each serial port must have an I/O address, and an interrupt (IRQ). There are the four serial ports corresponding to COM1 - COM4 (on Windows):

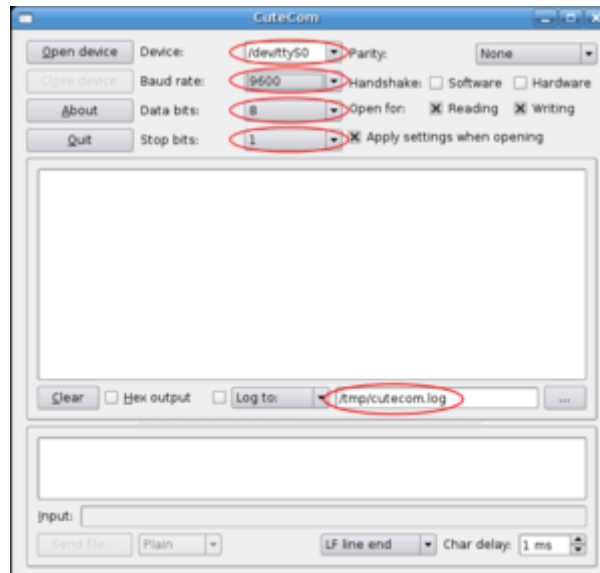
```
/dev/cua0 | /dev/ttyS0 (COM1) address 0x3f8 IRQ 4
/dev/cua1 | /dev/ttyS1 (COM2) address 0x2f8 IRQ 3
/dev/cua2 | /dev/ttyS2 (COM3) address 0x3e8 IRQ 4
/dev/cua3 | /dev/ttyS3 (COM4) address 0x2e8 IRQ 3
```

School computers use **/dev/ttyS0** as the standard serial interface. In case, you are working with a USB adapter use ttyUSB0 instead.

**Note for your own Linux:** If Linux does not detect any serial ports when booting, then make sure that serial support is compiled and enabled in the kernel. The **/dev/ttySX** devices are for incoming connections whereas **/dev/cuaX** devices are for outgoing connections. “X” represents the serial port number. In this document refers *COM1 to ttyS0, COM2 to ttyS1, COM3 to ttyS2, and COM4 as ttyS3*. Any references given to a specific device in /dev (directory where the device nodes are located) will always prepends /dev to avoid confusing. Notice that by default these devices have overlapping IRQs. You cannot use all of the ports in this default configuration, and you must reassign different IRQs.

## cuteCom

CuteCom is a serial-terminal tool for “Windows” user working with Linux due to its graphic user interface, which is straight-forward if you are not used to the console. Change the value by tipping the correct value after a click on the appropriate input field. To start the tool, open a terminal and type the command `cuteCom&` followed by [ENTER]. Check the device node that must be a `ttyUSB0` node for an USB adapter.



## minicom

Open a terminal as explained. Start the serial-terminal tool **minicom** by typing the command `minicom`. If this does not work, check if the tool is installed. You can use tools like “which” or “find” in case you work with “apt” as package manager, install minicom with root rights by using the appropriate command.

```
which minicom
```

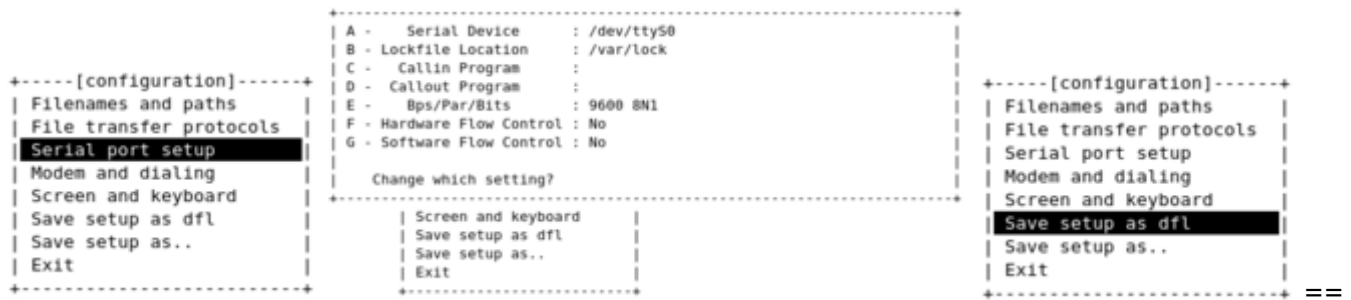
```
sudo apt-get install minicom
```

If you have any problems with the installation, contact the system administrator or the responsible lecturer. If **minicom** stops by a permission error when attaching a communication node, start it with the option **-s** for entering directly into the set-up dialogue. Check and adapt settings if they differ from those shown in the figures shown below.

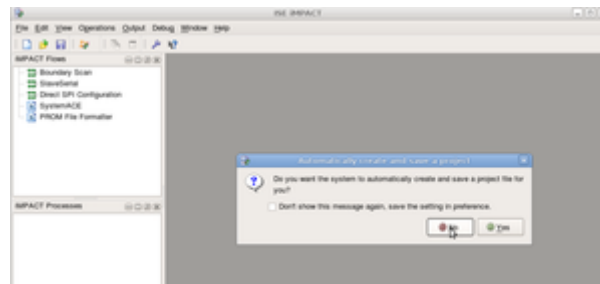
```
minicom -s
```

The configuration process is described briefly below:

- Start the tool with the command **minicom**
- With the shortcut [ctrl]+[A], you are able to switch to the interaction mode
- With the shortcut [ctrl]+[A] and [Z], you can open the help dialogue
- With the shortcut [ctrl]+[A] and [O], you will open the configure dialogue
- Open the interface configure dialogue (see in figure below)
- Make the same configuration as shown in figure below
- Save the settings as default values
- For closing minicom use [ctrl]+[A] followed by [Q]

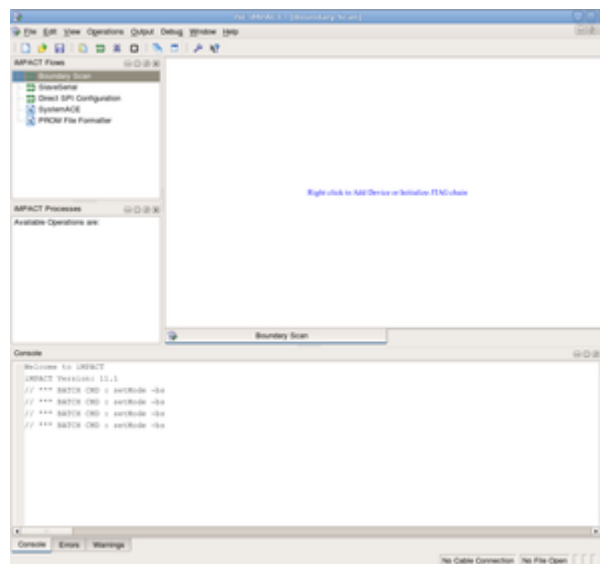


## How to download a bit file by using iMPACT

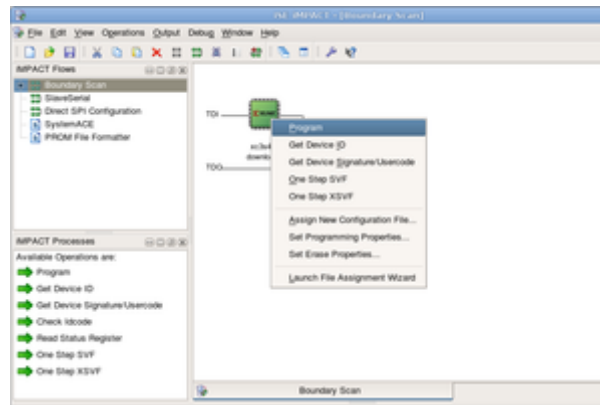


At that point say no for an automated saving of the project expect you are working in the corresponding exercise source folder.

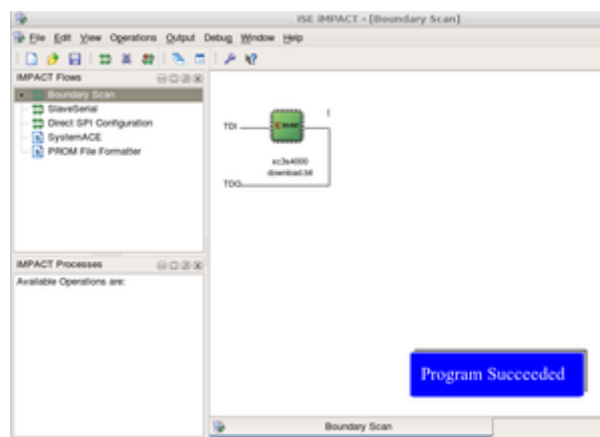
Make no new project because there is no need for doing this. It is required in case of real embedded projects in order to save necessary time.



Make sure the device is connected correctly. If every thing is checked carefully, make a boundary scan. The simplest way to do so is by click on "Initialize Chain" (shortcut is [Ctrl] + I). Assign a configuration file by browsing to the directory called implementation in the appropriate project tree. If you get there select the file named **download.bit**.



The FPGA icon was green before having a configuration file assigned. Right click on the device (FPGA icon) this changes the color of the icon back to green. Select “Program” in the pop-up menu.



If the programming result is **“Program Succeeded”** every thing is ready for further debugging or verification.

Just in brackets. If you wanna use a nice feature of Linux proceed a cat on the device node for getting the transmitted data over RS232. The command to use is

```
cat /dev/ttyS0
```

This is just in case you wanna check the data without a serial-terminal by a simple file IO.

## Conclusion

The “Base System Builder” can be used in XPS to create a project. Several files including an MHS file - representing the processor system - are created. A “System Assembly View” - representing the hardware system - provides the hardware system parameters. After the system has been defined, the net-list of the processor system can be created. In future labs, you will learn how to add additional IP-cores and simulating the design. Not only have you got an overview to the EDK design-flow but you have seen how to work with real hardware.

## Solutions

- [Lab1 design as tar archive](#)

# Lab2: Extend a Hardware Design

## Introduction

This exercise guides you through the process of adding additional IPs to an existing SoC project by using Xilinx Platform Studio (XPS). Furthermore, you learn also how to use the IP catalog tab that is the best way for browsing the available cores. At the end of the exercise, you will create the design net-lists and implement the design by using the ISE design-flow for testing the design on the target.

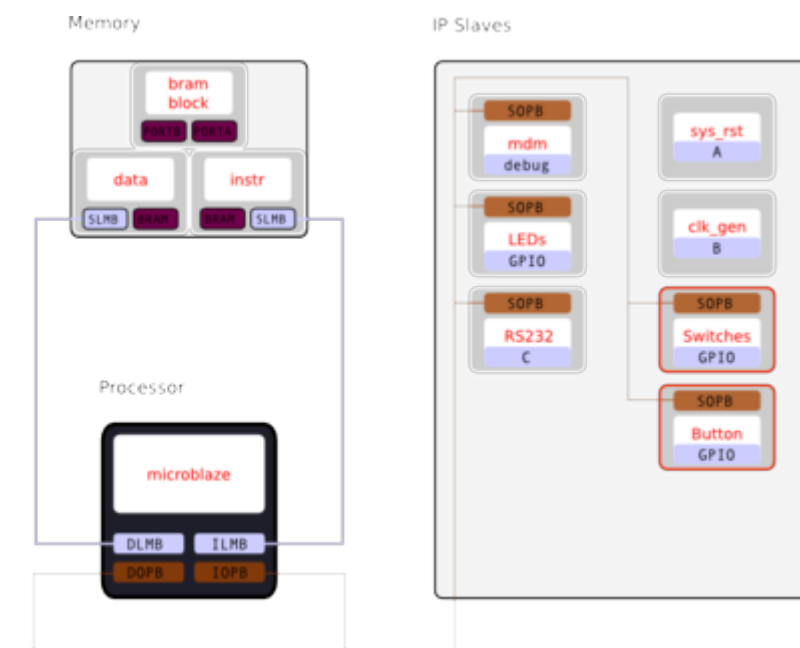
## Objectives

After working out the first lab, you will be able to:

- Add additional IPs to a hardware design
- Understand the UCF and MHS files
- Furthermore, you will know where to get information from

## Procedure

The purpose of this exercise is to extend the hardware design started by the previous exercise. Lab one included the MicroBlaze as processor, a couple of peripheral IP cores such as a RS232 or a GPIO core for the LEDs. Conversely to older versions of Xilinx's design suite the actual version uses for the inter-component communications the SPLB as bus system. However, the second exercise adds another GPIO core to extend the hardware design for accessing the push buttons. You will use the dialogue mode of the XPS and the text mode features for adding the following IPs



- GPIO over PLB to interface the push buttons and dip switches on the GECKO3main

You will also analyze the file **system.mhs** for getting a better overview about the various hardware specifying sections.

This lab comprises four primary steps: Firstly, you will add an IP to the SoC project of the previous chapter. Secondly, will the hardware system be extended by binding in the added cores. Furthermore, through analysing the MHS file you will get some information out about the project you are working on. Last but not least, by following some guidelines, how to use the “Project Navigator”, you learn how to implement the design. After completing all steps carefully you will add a software application (will be run on the softcore processor), generate the bitstream, download on the FPGA and verify the new functionality.

## Adding IP to the Processor System

Create a folder named e.g. lab2 and copy the contents of the lab1 folder into the new created folder if you wish to continue with the design created during the previous exercise. Launch “Xilinx Platform Studio (XPS)” and browse to the new project folder and open the **system.xmp** file located in

```
/home/<USERNAME>/SoC/Lab2/system.xmp
```

Otherwise, do every single step of the previous exercise, first.

Short step-by-step process-flow:

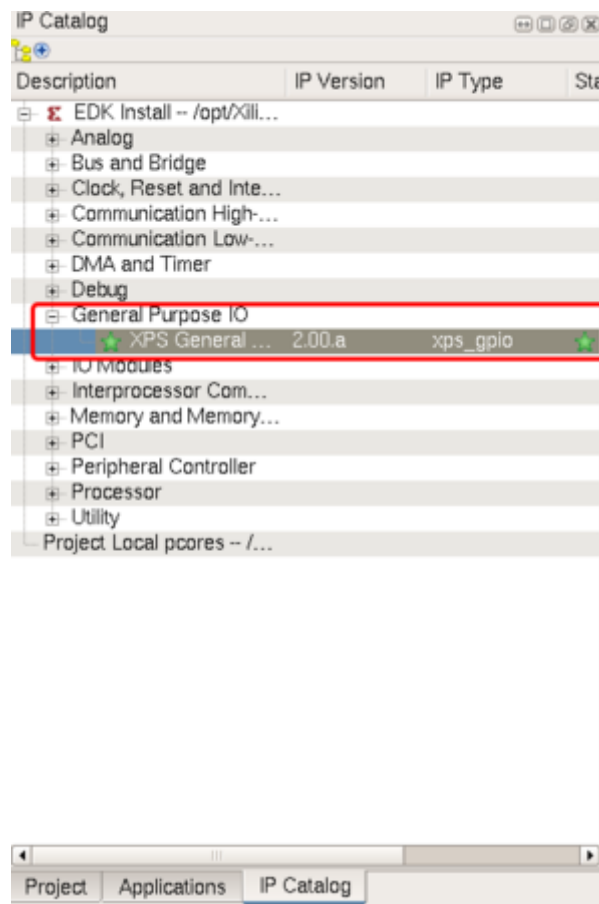
- Open a terminal and change to your workspace by using **cd**. Being in the directory **/SoC**, create a new folder containing the files from exercise one. This is feasible by a recursive copy of the lab one folder. Use for this the command

```
cp -Rf lab1 lab2
```

- Afterwards change into the new directory and control the contents
- Now you have to start a new XPS session with the command **xps&**
- Select **File → Open Project** and browse to either the **/SoC/lab2** folder or to the one where the files are.
- Select **system.xmp**, which contains the project information
- After having the project opened it is required to clean it under **Project → Clear all generated files**

## Extending the Hardware System

Add the **XPS GPIO** IP core to the processor system using “System Assembly View” panel.



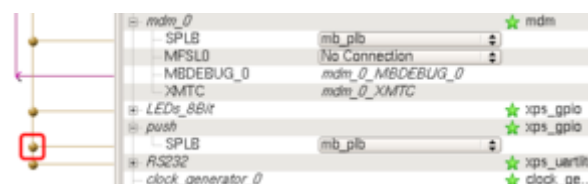
XPS provides two methods for adding peripherals to an existing project. You will use the first method, - the System Assembly View panel -, to add and connect most of the additional IPs. The second method is editing the MHS file and not requested for beginners. However, it is handy for system debugging.

- Select “IP Catalog” tab in the left window and click on the plus sign - next to the “General Purpose IO” entry to view the available cores.
- Double-click on the “XPS Generale Purpose IO” core (version 2.00.a) once, to add one instances to the System Assembly View.
- Change the instance name of the peripheral to push, by clicking once in the name column.

At this point the peripherals tab should look like shown below.



- Click in “Bus Connection” column for the push instance to connect them as a slave devices to the SPLB. Afterwards, the connection changes from unfilled to filled brown circle.



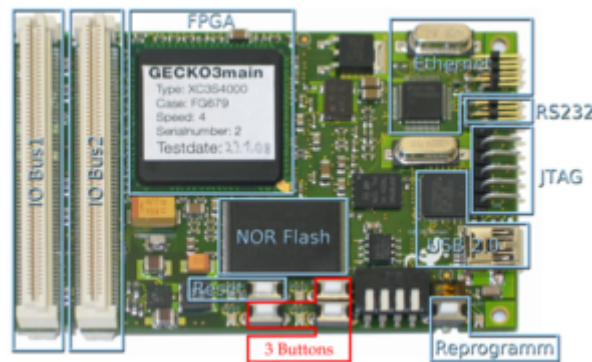
- Select the addresses filter. You can manually assign the base address and size of the peripherals or have XPS generate the addresses for you.
- Click the button “Generate Addresses” to automatically map the peripheral to the right addresses.

The base address and high address will change as shown below.

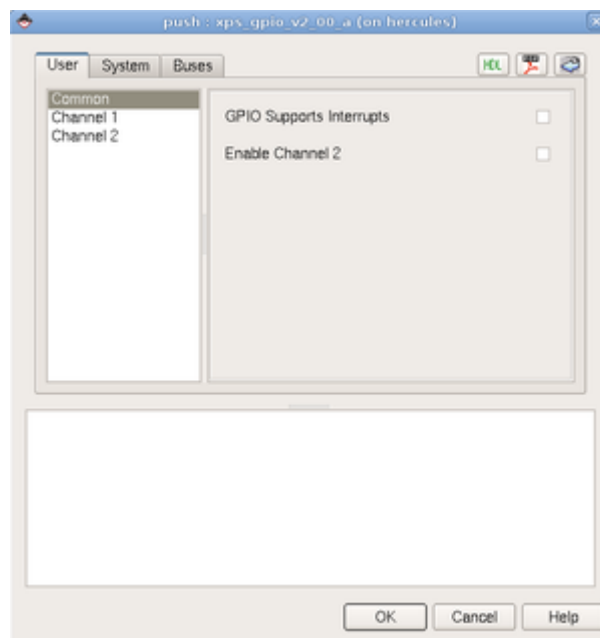
Instance	Base Name	Base Address	High Address	Size	Bus Interface	IO IP Type	IP Version
inst0_000	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_000 2.10.0
inst0_001	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_001 2.10.0
inst0_002	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_002 2.10.0
inst0_003	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_003 2.10.0
inst0_004	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_004 2.10.0
inst0_005	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_005 2.10.0
inst0_006	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_006 2.10.0
inst0_007	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_007 2.10.0
inst0_008	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_008 2.10.0
inst0_009	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_009 2.10.0
inst0_010	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_010 2.10.0
inst0_011	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_011 2.10.0
inst0_012	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_012 2.10.0
inst0_013	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_013 2.10.0
inst0_014	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_014 2.10.0
inst0_015	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_015 2.10.0
inst0_016	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_016 2.10.0
inst0_017	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_017 2.10.0
inst0_018	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_018 2.10.0
inst0_019	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_019 2.10.0
inst0_020	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_020 2.10.0
inst0_021	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_021 2.10.0
inst0_022	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_022 2.10.0
inst0_023	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_023 2.10.0
inst0_024	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_024 2.10.0
inst0_025	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_025 2.10.0
inst0_026	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_026 2.10.0
inst0_027	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_027 2.10.0
inst0_028	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_028 2.10.0
inst0_029	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_029 2.10.0
inst0_030	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_030 2.10.0
inst0_031	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_031 2.10.0
inst0_032	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_032 2.10.0
inst0_033	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_033 2.10.0
inst0_034	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_034 2.10.0
inst0_035	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_035 2.10.0
inst0_036	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_036 2.10.0
inst0_037	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_037 2.10.0
inst0_038	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_038 2.10.0
inst0_039	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_039 2.10.0
inst0_040	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_040 2.10.0
inst0_041	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_041 2.10.0
inst0_042	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_042 2.10.0
inst0_043	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_043 2.10.0
inst0_044	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_044 2.10.0
inst0_045	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_045 2.10.0
inst0_046	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_046 2.10.0
inst0_047	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_047 2.10.0
inst0_048	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_048 2.10.0
inst0_049	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_049 2.10.0
inst0_050	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_050 2.10.0
inst0_051	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_051 2.10.0
inst0_052	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_052 2.10.0
inst0_053	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_053 2.10.0
inst0_054	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_054 2.10.0
inst0_055	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_055 2.10.0
inst0_056	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_056 2.10.0
inst0_057	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_057 2.10.0
inst0_058	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_058 2.10.0
inst0_059	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_059 2.10.0
inst0_060	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_060 2.10.0
inst0_061	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_061 2.10.0
inst0_062	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_062 2.10.0
inst0_063	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_063 2.10.0
inst0_064	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_064 2.10.0
inst0_065	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_065 2.10.0
inst0_066	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_066 2.10.0
inst0_067	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_067 2.10.0
inst0_068	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_068 2.10.0
inst0_069	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_069 2.10.0
inst0_070	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_070 2.10.0
inst0_071	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_071 2.10.0
inst0_072	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_072 2.10.0
inst0_073	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_073 2.10.0
inst0_074	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_074 2.10.0
inst0_075	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_075 2.10.0
inst0_076	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_076 2.10.0
inst0_077	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_077 2.10.0
inst0_078	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_078 2.10.0
inst0_079	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_079 2.10.0
inst0_080	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_080 2.10.0
inst0_081	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_081 2.10.0
inst0_082	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_082 2.10.0
inst0_083	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_083 2.10.0
inst0_084	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_084 2.10.0
inst0_085	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_085 2.10.0
inst0_086	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_086 2.10.0
inst0_087	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_087 2.10.0
inst0_088	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_088 2.10.0
inst0_089	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_089 2.10.0
inst0_090	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_090 2.10.0
inst0_091	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_091 2.10.0
inst0_092	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_092 2.10.0
inst0_093	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_093 2.10.0
inst0_094	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_094 2.10.0
inst0_095	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_095 2.10.0
inst0_096	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_096 2.10.0
inst0_097	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_097 2.10.0
inst0_098	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_098 2.10.0
inst0_099	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_099 2.10.0
inst0_100	C_BASEADDR	0x00000000	0x0000FFFF	64K	2	SLIM0	inst0_100 2.10.0

There are three push buttons on the GECKO3main board. You will first configure the push instance according to the following parameters, and then set the data port of the instance as external.

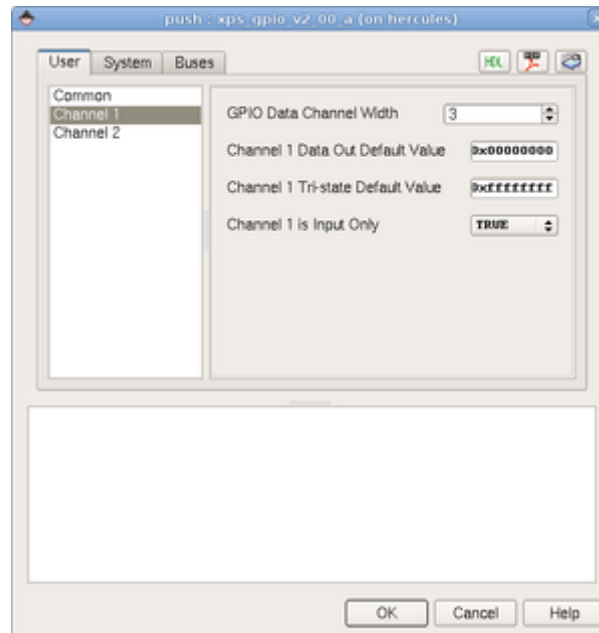
- Select the Ports filter in the tool-bar of the **System Assembly View**.
- Double-click on the push instance to open the configuration window.
- Notice that the peripheral can be configured for two channels, but since we want to use only one channel keep **Enable Channel 2** unchecked.
- Click on the **GPIO Data Bus Width** of channel one set it to 3 due to the board has three push buttons.
- The settings for the common parameters should be set according to figure below.



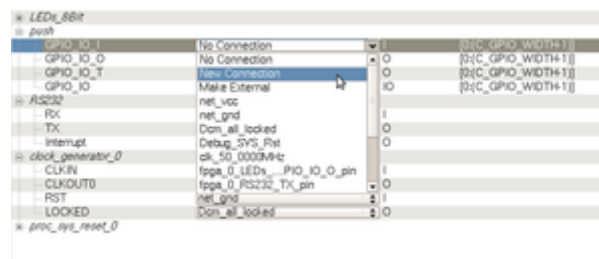
Next, set channel one as a input channel only. Per default a channel is a bi-directional communication element to the outside world.





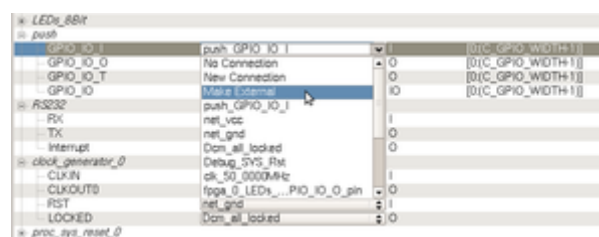


Click in the column Net (port view) of our added push core to set a appropriate port name for *GPIO\_IO\_I*. As for activating, hit the option new connection first. After doing it you are free to rename the port, but generally there is no need to do so. The port's name is probably *push\_GPIO\_IO\_I*, which is a good description of the port and its function. Get some additional information from the figure below.



It is important to declare the port as an external port. It must be an external port so make sure no error happen.

External ports are accessible from outside of the FPGA design. That is why this port need to be external.



Finally bind the port by the UCF description to the correct FPGA pin. The next step is for checking any changes made until now. Switch to the external ports view (straight at the top of the port assembly view) there should be an external accessible port (as 3 bit vector) **push\_GPIO\_IO\_I\_pin**.

Name	Net	Direction	Range
<b>External Ports</b>			
fpga_0_RS232_RX_pin	fpga_0_RS232_RX_pin	2 I	5
fpga_0_RS232_TX_pin	fpga_0_RS232_TX_pin	2 O	5
fpga_0_LEDs_8Bit_GPIO_1...	fpga_0_LEDs_8Bit_GPIO_1...	2 O	5 [0:7]
fpga_0_clk_1_sys_clk_pin	don_clk_5	2 I	5
fpga_0_rst_1_sys_rst_pin	syn_rst_5	2 I	5
<b>push GPIO IO 1 pin</b>	<b>push GPIO IO 1</b>	<b>2</b>	<b>5 [0:2]</b>
<b>microblaze_0</b>			
+ dmb			
+ imb			
+ mb_pib			
+ dmb_cndr			
+ imb_cndr			
+ imb_brn			

Now every thing should be ready for building the libraries. Libraries generated by the **XPS** provide an interface to access the hardware by software over a register based interface. After compiling the libraries, add an existing C program to implement some functionalities for push buttons and LEDs. Once you compile the program, change the project option to generate the processor system. This module can be included as a top-level module in a **ISE** project.

No-worries if you do not know how to do so. The next few steps guide you through the process safely.

1. Click on or **Software → Generate Libraries and BSPs** to generate all libraries and necessary header files (\*.h). This is an important step because the system will consider all hardware changes. At the end have a look into the applications window next to the IP catalog.
2. Click on Add Software Application Project, name it for example as myTest.
3. Copy the sources and header files as well as the linker script from **TestApp\_Peripheral\_microblaze\_0** to **myTest** recursively. Having done so, create a new file **lab2\_skel.c** (or **lab2\_11.c** hence the 11 indicates the tool-chain's version the file has been successfully tested on.) in **myTest** and copy the content below into it (or just download the file below to **myTest**). Import all the necessary files into your XPS project (Click on "Sources" and "Headers" in the Application Tab in XPS). You can find out which files are necessary by examining their code or reading their name.
4. Set the new SW project as active, now. The option to tick is "Mark to Initialize BRAMs". (Right click on the appropriate project for having this option). If needed have a look in the figures below. That results in copying the byte-code into the BRAM.

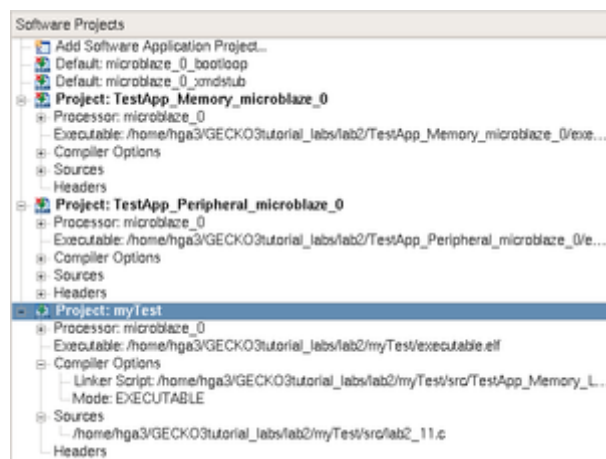
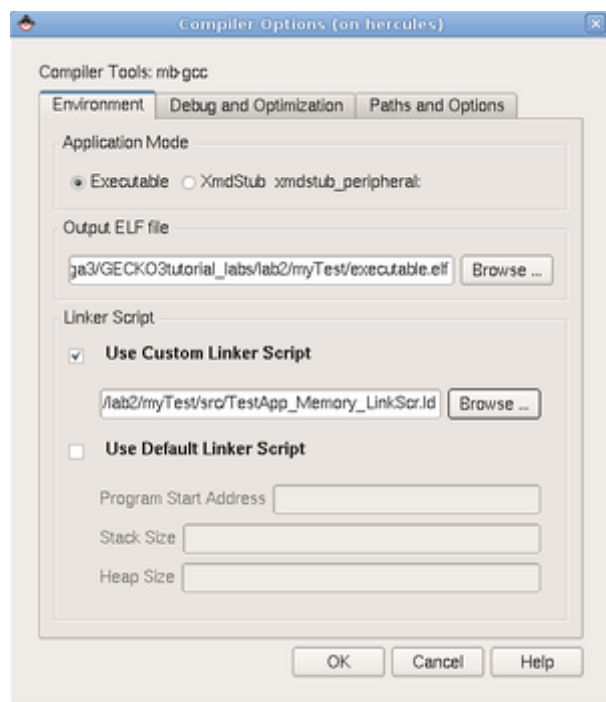
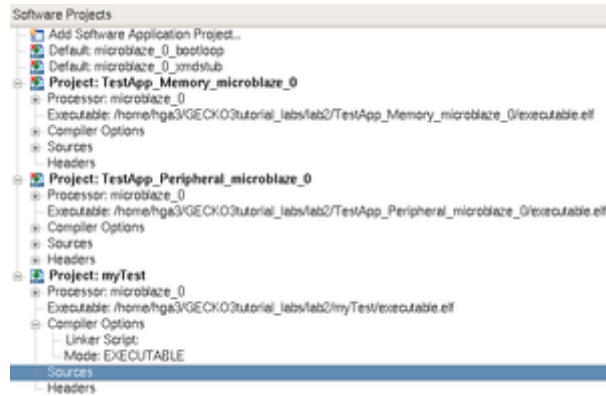
### lab2\_skel

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"
int main ( void )
{
    XGpio push;

    volatile int i;
    int push_state, ERR_push, ERR_dipi;

    xil_printf("-- Start of the Program --"); // print to the serial interface
    ERR_push = XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Initialize the port
    while( ERR_push == XST_DEVICE_NOT_FOUND)
    {
        ERR_push = XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Initialize the port
        xil_printf("ERROR : initialisation of buttons not succeeded\n");
    }
    XGpio_SetDataDirection(&push, 1, 0xffffffff); // Initialize the port as input
    xil_printf("Initialisation succeeded ...\n");

    while(1) // endlessloop
    {
        push_state = XGpio_DiscreteRead(&push, 1); // read the Buttons
        xil_printf("Push Button State: 0x%x \r", push_state);
        for( i = 0; i < 999999; i ++); // wait cycle
    } // end while
} // end main
```

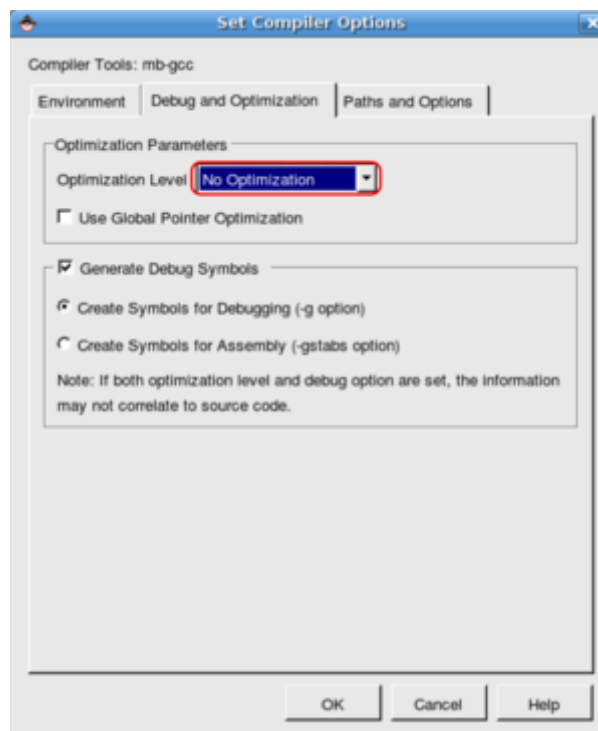


At the end of the code file is a busy loop for realizing a wait cycle. On that level and for this specific topic it is the best practice working with a busy loop. When you now compile the source, the loop will be optimized away

by the compiler. To prevent the compiler of doing this use the key word volatile or change the optimization level to "-O0".

**Volatile:**\* The reason for having this type qualifier is mainly to deal with the problems that are encountered in real-time or embedded systems programming using C. Imagine that you are writing code that controls a hardware device by placing appropriate values in hardware registers at known absolute addresses. However, a major problem with previous C compilers would be in a loop which tests a register or does something similar. Any self-respecting optimizing compiler would notice that the loop tests the same memory address over and over again. It would almost certainly arrange to reference memory once only, and copy the value into a hardware register, thus speeding up the loop. This is, of course, exactly what we don't want; this is one of the few places where we must look at the place where the pointer points, every time around the loop. For this reason it is important to declare the loop variable as a volatile.

**Compiler optimization:** In the Application tab, double-click on compiler options and set the optimization to "No Optimization" (Fig. below). This will ensure that the for loop (used for realizing a software delay) in the source code is not optimized away (this is not required comparing to the solution with the key word volatile).




## Edit the UCF file

Before you have finished, make the IO description in the UCF. To do this double click on the system. UCF file located in the project tab and add the following code to assign pins to the push button IP core. The source shown in the listing below is needed (lab2\_11.ucf). If you are interested in a short overview over the UCF.

[lab2\\_11.ucf](#)

```
# ##### Module Push_Button_3Bits constraints
NET push_GPIO_I0_I_pin<0> LOC = E7 | IOSTANDARD = LVCMOS33 | PULLUP; #
NET push_GPIO_I0_I_pin<1> LOC = G10 | IOSTANDARD = LVCMOS33 | PULLUP; #
NET push_GPIO_I0_I_pin<2> LOC = E12 | IOSTANDARD = LVCMOS33 | PULLUP; #
```

Save the \*.ucf file and Click on  to compile the source code. Make sure that it compiles error free.

## Analyzing the MHS File

Double click the system.mhs file in the project bar to open it.

**Study the external ports sections in the MHS file and answer the following questions:**

- ☐ Number of external ports
- ☐ Number of external ports that are output
- ☐ Number of external ports that are input
- ☐ Number of external ports that are bidirectional

**List the instances to which the system clock is connected**

- ☐ Exercise checked by lecturer

**List the devices connected to the PLB**

- ☐ Exercise checked by lecturer

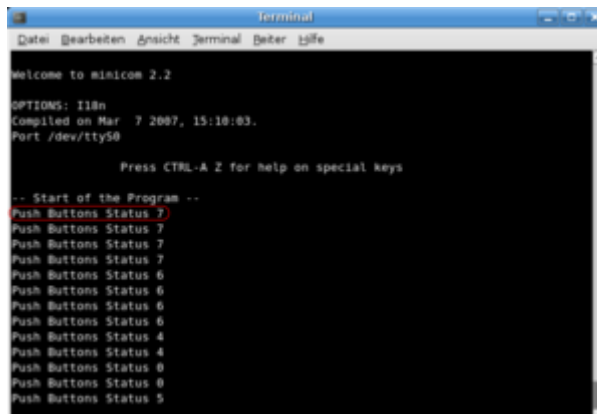
**Draw the address map of the system, providing instance names (use the filter Addresses)**

- ☐ Exercise checked by lecturer

## Verify in hardware

Start a serial-terminal session as you learned during the previous lab exercise. Connect and power up the GECKO3EDU system. Afterwards, update the project under **Device Configuration → Update Bitstream**. This may take a few minutes to synthesis, implement, and generate the bitstream. Download the bitstream (under Device Configuration → Download Bitstream when working completely local or by the iMPACT starter when working remotely).

Once the bitstream is downloaded, you should see an output displayed in the serial-terminal similar to the one in the figure below.



```
terminal
Datei Bearbeiten Ansicht Terminal Fenster Hilfe

Welcome to minicom 2.2

OPTIONS: I18n
Compiled on Mar  7 2007, 15:10:03.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys

... Start of the Program ...
Push Buttons Status 7
Push Buttons Status 7
Push Buttons Status 7
Push Buttons Status 7
Push Buttons Status 6
Push Buttons Status 6
Push Buttons Status 6
Push Buttons Status 6
Push Buttons Status 6
Push Buttons Status 4
Push Buttons Status 4
Push Buttons Status 0
Push Buttons Status 0
Push Buttons Status 5
```

You can see that **button\_0** to **button\_2** are connected as pull-ups. In consideration of the fact that all buttons

must be pushed to get a zero as output a exercise is introduced.

**Change the observed behavior by two different methods:**

- ☐ By a software solution (C)
- ☐ By a hardware solution (VHDL)

How to create your own IP core will be a subject in a future exercise. Therefore, do not concern if you feel like missed something.

Disconnect and close the serial-terminal window session.

## Additional exercise

During this exercise you learned how to add an GPIO IP core. Now it is high time to check what you learned. Show how to add an additional GPIO IP core for having the switches accessible by the software. To avoid weirdness some brief information is given so make sure you understand them first.

**Implement the switches:** To have some help, follow the next steps:

1. Add the GPIO IP core (change the name to dip)
2. Connect the core to the PLB
3. Set the correct bus-size and data transfer direction.
4. After configure the IP-core, edit the system.ucf file to end-up with a correct pin-assignment. Additional information can be found on the [website](#).
5. In the C file you have to initialize the GPIO and to add a **xil\_printf()** for a correct output on the serial terminal
6. Last but not least, update the addresses and all the libraries, before updating the project.

If you need any help talk with other students, first. If there is no solution consult the solution files located at the bottom of this exercise.

If there is any problem check the address-mapping, first. The code section must start at the begin due to **MicroBlaze's** reset and initialization behavior.

## UCF overview

The UCF (User Constrains File) is an ASCII file specifying constraints on the logical design. Create this file and enter your constraints with any text editor, also use the Constraints Editor to create constraints within a UCF file. These constraints affect how the logical design is implemented in the target device. You can use the file to override constraints specified during design entry. To get an impression about the UCF syntax study briefly the following examples. This are just examples so you have to adapt the settings to your specific needs.

### ucf\_example

```
# Clock definition:
NET CLOCK LOC = AA12;
NET CLOCK TNM_NET = "CLOCK_50";
TIMESPEC TS_CLOCK_50 = PERIOD CLOCK_50 50 MHz HIGH 50 %;

#Time invariant assignment:
```

```
NET reset LOC = D8;
NET reset TIG;

#Pin assignment:
NET LED LOC = W2;
NET SWITCH_n LOC = D1;
NET VGA_RED<2> LOC = X4;
NET VGA_RED<1> LOC = W5;
NET VGA_RED<0> LOC = T9;

#Output driver description:
NET my_netname FAST;
NET my_netname DRIVE = 12;
NET " *" IOSTANDARD = LVTTTL;

#Pullup/-down:
NET netname PULLUP;
NET netname PULLDOWN;

#Combination of IO descriptions:
NET netname LOC = A1 | DRIVE = 2 | FAST;
```

If you need more information or some answers to a specific problem consult Xilinx's website. The following link may help you by some problems:[Xilinx documentation](#)

## MHS overview

The Microprocessor Hardware Specification (MHS) file defines the hardware component. The MHS file serves as an input to the Platform Generator (Platgen) tool. An MHS file defines the configuration of the embedded processor system, and includes the following:

- Bus architecture
- Peripherals
- Processor
- System Connectivity
- Address space

## Conclusion

Xilinx Platform Studio (XPS) can create a MHS file representing the processor system. The system configuring by using peripheral parameters and controlling internal and external ports. After defining the system, the processor system net-lists can be created.

In further labs in this course, you will learn how to add user specific elements, add software to the system, simulate the design, debug the software, and verify the functionality of the completed design.

## Solutions

- [Lab2 extended design as tar archive](#)

# Lab3: Adding a Custom IP

## Introduction

This lab guides you through the process of adding a custom peripheral to a processor system by using the **Create and Import Peripheral** wizard. You will connect the IP with help of the IPIF to the PLB (Processor Local Bus).

## Objectives

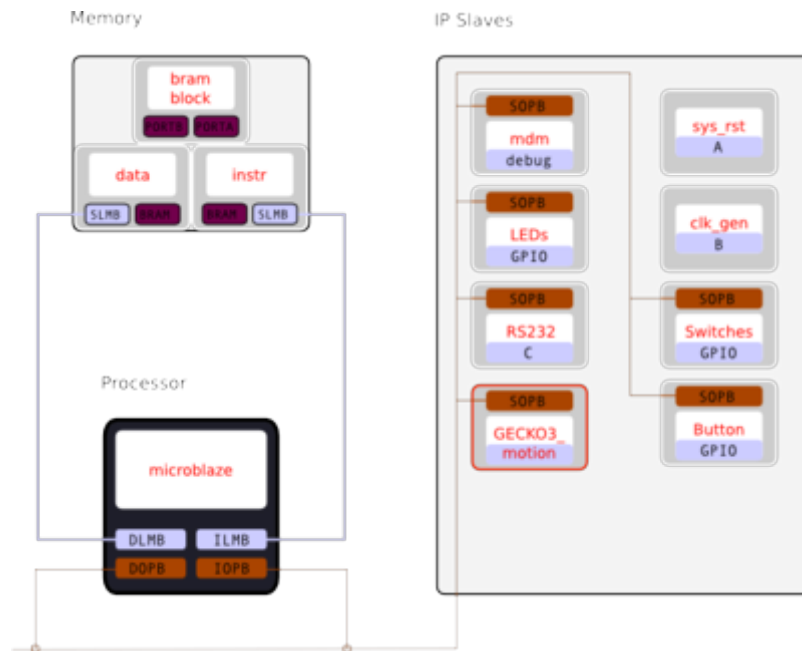
After working out the first lab, you will be able to:

- Create a custom peripheral and import it to the IP catalog
- Add the custom peripheral to your design
- Modify the UCF file
- Generate the bit-stream and verify the functionality in hardware
- Write a driver for the IP

## Procedure

The purpose of this exercise is to complete the hardware design started in the first lab and extended in the previous one. The first exercise included the MicroBlaze, a debug module, an UART core, LMB BRAM controller, and LMB BRAM. The second exercise added two instances of a GPIO cores. During this exercise you are going to learn how to add a custom IP to the SoC design. The custom IP will be connected over the IPIF, a simple and easy to use interface for interacting over a intelligent system bus e.g. PLB. The platform GECKO3EDU has interesting components and interfaces that you are going to use during this exercise. At the end of the exercise you will know how to use and control the motors of the GECKO3robot. It is simple writing a PWM core for the motors but it is an easy understandable examples that focuses on how to add a custom IP. After completing the exercise you will be able to add more complex custom cores or how modified them to your individual needs, respectively. A easy way of adding custom designs is to work with the wizard function **Create and Import Peripheral** wizard, integrated in Xilinx Platform Studio (XPS). To do so create a user peripheral from an HDL module, add an instance of the imported peripheral, and modify the system.ucf file to provide an interface to the motors on the GECKO3robot. The IP core gecko3motion is the motor controller. Download the skeleton files later in this exercise.



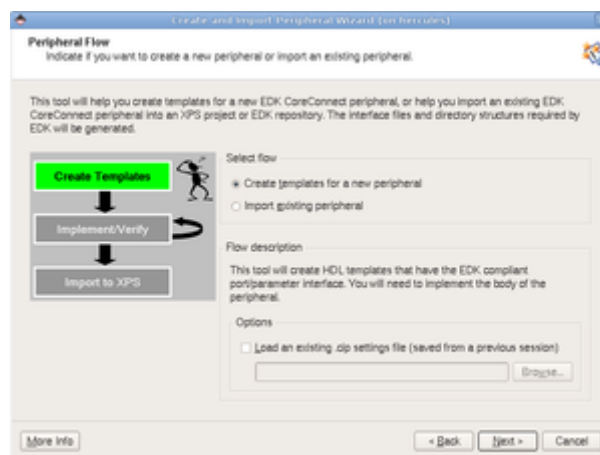


## Open the Project

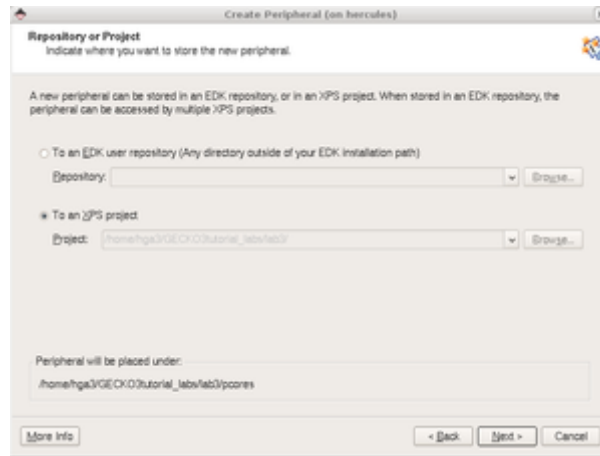
Create a lab3 folder and copy the contents of the lab2 folder into it. If you wish to continue with the design you created during the previous lab otherwise start from scratch. After copying, open the project in XPS. If you need more detailed information have a look at the descriptions of lab2 or lab1.

## Create a Custom Peripheral

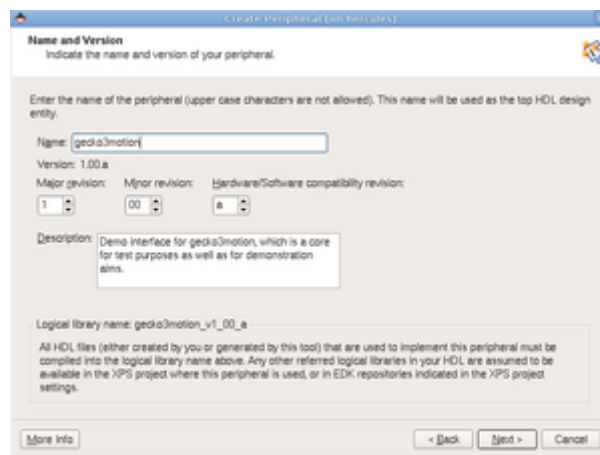
In XPS, use the “Create or Import Peripheral” wizard to create an HDL template for integrating a custom controller into the SoC design. To open the wizard in XPS, select **Hardware → Create or Import Peripheral**. Afterwards click “Next” to continue the “Create and Import Peripheral Wizard”. Get some inspiration in the figures below 😊.



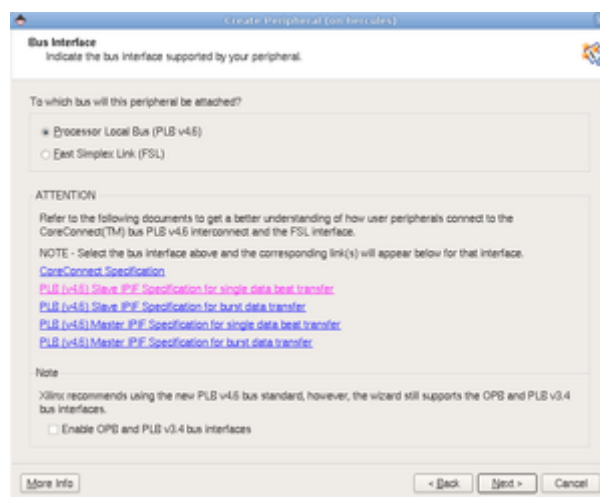
In the “Peripheral Flow” panel select “Create templates for a new peripheral” and click “Next”. The option “To an XPS project” should be selected as default. Make sure the path is pointing to the correct directory, change it otherwise.



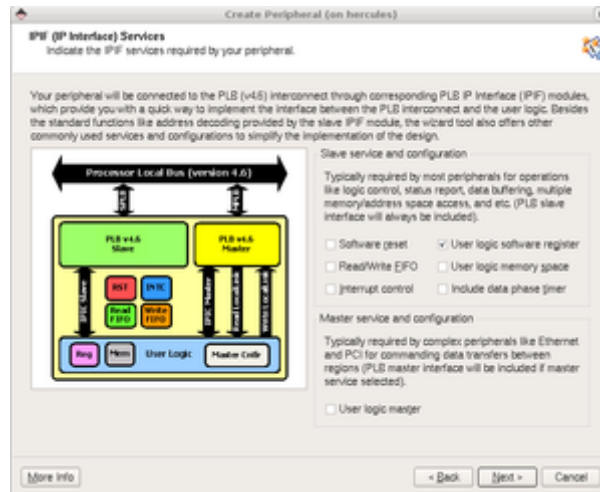
Click “Next” and enter **gecko3motion** in the name field and click “Next”.



In the bus interfaces panel, select Processor, and click “Next” as shown below.



Next, un-check the option “include data phase timer” as shown in figure below. At this point, the option will be deactivated because there is no need for such a feature in the gecko3motion IP.

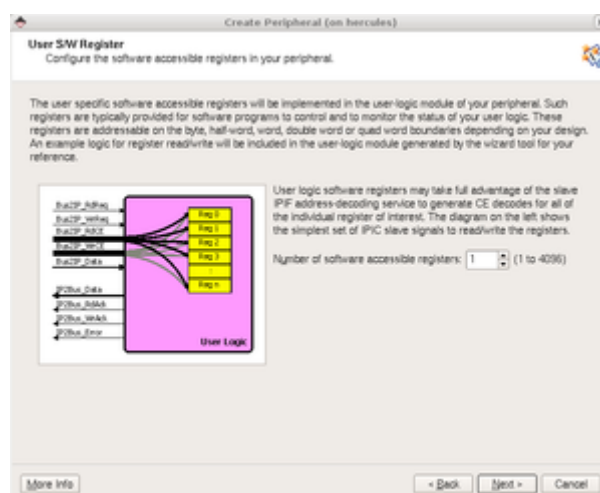


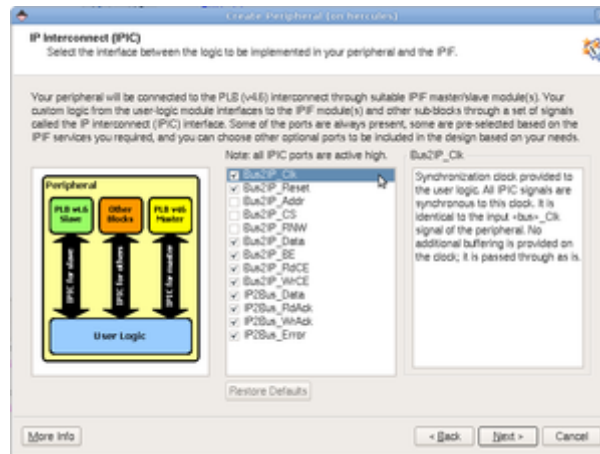
Continue with the wizard. Find the option for burst and cache-line support, next. Let this support disabled because there is no need for this feature at the moment.



By the following panel choose the quantity of software accessible registers. For this example only one register is required. Again, click next and chose the external ports if necessary.

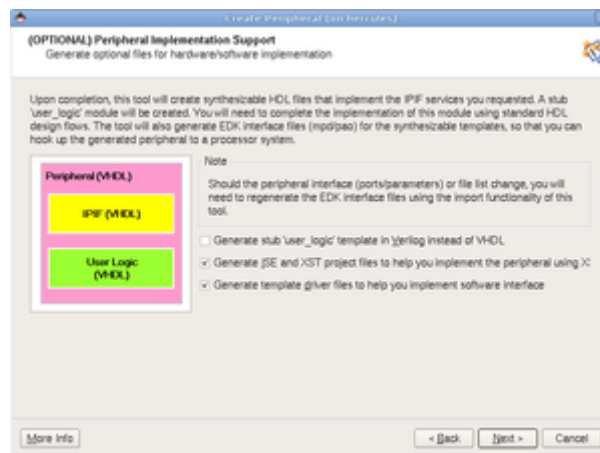
Read all port descriptions carefully. In case you do not remember the description a little while later check the description file located in the IP folder structure.





In the (OPTIONAL) Peripheral Simulation Support panel, leave “Generate BFM simulation platform” disabled, and click Next. Enable “Generate ISE and XST project files” and “Generate template driver files” and click “Next”, afterwards.

Click Next, until the template creation process has come to an end. Click on IP Catalog tab in XPS and observe that GECKO3motion IP core exists in the project repository, now.



Sometimes a refresh of the project is required. Refresh the project by clicking on the refresh button .

## Create the motor controller core in VHDL

Follow these steps given to create the motor controller core:

First of all, think about a motor controller briefly. Some questions that help designing a motor controller:

- How many motors does the system contain?
- How to control them? (PWM)
- How are the motors connected to the FPGA?
- Which HW blocks are involved?

After thinking about this questions you will end up with a concept similar to the one described by the tutorial in the steps following next.

- The robot has two motors each controlled by a H-bridge.
- Due to the H-bridge we can use a PWM.
- There are two signal lines for each motor (e.g. mot\_1A and mot\_1B).
- Create a main instance “the controller” the controller itself includes two pwm modules, due to

the two motors.

- For a certain direction the motors turn in opposite direction. Therefore the PWM controller

includes a direction flag or the value given to the PWM controller will be adjusted to match the requirements, before.

- We end up with two blocks a PWM block and a motor controller, respectively.

At that point a block diagram helps. As for exercising draw the block diagram that shows the motor controller that is divided in several blocks as mentioned above.

Download and store the source template given below into the system tree. Follow the instruction given, next:

- Select "File→New". This opens a new text document that we use for the PWM description

written in VHDL.

- Copy and paste the following code (listing 3.1) into the new text document and save it named as pwm.vhd in the folder **<YOUR TEMPLATE NAME>/hdl/vhdl**, afterwards.

pwm.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-----
entity pwm is
    generic (CNT_SLOW_RANGE : INTEGER := 500); -- f = 50MHz / 500 = 100kHz
    port ( i_clk: in  STD_LOGIC;                -- 50MHz clock
          i_pwm: in  INTEGER RANGE 0 TO 255;    -- duty-cycle
          o_pwm: out STD_LOGIC);               -- binary PWM signal
end pwm;
-----
architecture Behavioral of pwm is
begin
    PROCESS(i_clk) -- essential for if()
        VARIABLE v_cnt_slow: INTEGER RANGE 0 TO (2147) := 0;
        VARIABLE v_cnt_pwm:  INTEGER RANGE 0 TO (2047) := (i_pwm*8);
    BEGIN
        IF (i_clk'EVENT AND i_clk='1') THEN

            if(v_cnt_pwm > 0) then
                v_cnt_pwm := v_cnt_pwm - 1;
            END IF;

            IF(v_cnt_slow >= CNT_SLOW_RANGE) THEN
                v_cnt_pwm := i_pwm * 8; -- LOAD pwm duty-cycle
                v_cnt_slow := 0;
            END IF;

            IF(v_cnt_pwm = 0) THEN
                o_pwm <= '0';
            ELSE
                o_pwm <= '1';
            END IF;
            v_cnt_slow := v_cnt_slow + 1;

        END IF;
    END PROCESS;
end architecture Behavioral of pwm;

```

```
end Behavioral;
```

- Repeat the steps described for a file named motor\_controller.
- Check, if the design matches the block diagram you have been drawing at the begin.

### motor\_controller.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--use IEEE.numeric_std.ALL;

--library gecko3motion_v1_00_a;
--use gecko3motion_v1_00_a.all;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity motor_controller is
    generic (SLOW_RANGE : INTEGER := 500);
    port(
        i_clk      : IN STD_LOGIC;
        i_nRst     : IN STD_LOGIC;
        i_period   : IN STD_LOGIC_VECTOR(7 downto 0);
        i_dir      : IN STD_LOGIC;
        i_run      : IN STD_LOGIC;
        o_M1A      : OUT STD_LOGIC;
        o_M1B      : OUT STD_LOGIC;
        o_M2A      : OUT STD_LOGIC;
        o_M2B      : OUT STD_LOGIC
    );
end motor_controller;

architecture behave of motor_controller is

    component pwm is
        generic (CNT_SLOW_RANGE : INTEGER := 500); -- f = 50MHz / 500 = 100kHz
        port ( i_clk: in  STD_LOGIC;               -- 50MHz clock
              i_pwm: in  INTEGER RANGE 0 TO 255;    -- duty-cycle
              o_pwm: out STD_LOGIC);               -- binary PWM signal
    end component;

    signal s_pwmM1 : std_logic;
    signal s_pwmM2 : std_logic;

begin

    MotorLeft : pwm generic map (
        CNT_SLOW_RANGE => SLOW_RANGE
    )
        port map (
            i_clk      => i_clk,
            i_pwm      => conv_integer(i_period),
            o_pwm      => s_pwmM1
        );

    MotorRight: pwm generic map (
        CNT_SLOW_RANGE => SLOW_RANGE
    )
        port map (
            i_clk      => i_clk,
            i_pwm      => conv_integer(i_period),
            o_pwm      => s_pwmM2
        );
end behave;
```

```

);

process(i_dir,i_run)
begin
    case std_logic_vector'(i_dir,i_run) is
        when "X0" => o_M1A <= '0';
                    o_M1B <= '0';
                    o_M2A <= '0';
                    o_M2B <= '0';

        when "01" => o_M1A <= '1';
                    o_M1B <= s_pwmM1;
                    o_M2A <= s_pwmM2;
                    o_M2B <= '1';

        when "11" => o_M1A <= s_pwmM1;
                    o_M1B <= '1';
                    o_M2A <= '1';
                    o_M2B <= s_pwmM2;

        when others => null;

    end case;
end process;

end behave;

```

## Modify the .PAO file

The .pao file contains a list of all source files that the peripheral contains. The list will be read in when running the “Peripheral Wizard” in import mode. Due to files has been added, modify the PAO file. There are several ways of proceeding. Follow the procedure described, next, to cover one method.

- Select “File→Open” and browse to the “pcores/gecko3motion\_v1\_00\_a/data” folder. Select the file “gecko3motion\_v2\_1\_0.pao” and click “Open”.
- At the bottom of this file find two lines: “lib gecko3motion\_v1\_00\_a user\_logic vhd1” and “lib gecko3motion\_v1\_00\_a gecko3motion vhd1”.
- Add the lines “lib gecko3motion\_v1\_00\_a pwm vhd1” and “lib gecko3motion\_v1\_00\_a motor\_controller vhd1” just above the two lines.
- Save all changes.

After the previous step the PAO file is ready to be used by the peripheral wizard. This time select “run in import mode”. Notice that the PAO file lists the source files in a hierarchical sensitive order. Therefore include them in consideration of hierarchical dependency. The top components are listed at the bottom of the PAO file. Also include entities before architectures, if you split them into separate files.

## Modify the Peripheral

During this step you are going to extend the created template for instantiate the **motor\_controller** core. For communication Reg0 is used. If a core is designed that is more like a communication device or simply a device for data transfer, use FIFOs. In this specific case, let the wizard generate a template without FIFOs therefore make sure this option is disabled.

Anyway, do the following steps for binding the controller into the template.

- Select from the menu “File→Open” and explore the project folder.
- Open the folders: “pcores/gecko3motion\_v1\_00\_a/hdl/vhd1”. This folder contains two source files that describe

the peripheral “gecko3motion.vhd” and “user\_logic.vhd”. The first file is the main part of the peripheral and it provides an interface to the PLB. The second file is where the custom logic will be included. This part is simplified by the “user\_logic.vhd” template file.

- Open the file “user\_logic.vhd”. A modification is needed, which instantiates the **motor\_controller** and connects it to the first slave register.
- Find the lines shown in the following code snipes. There add the code just below

“-USER <XY> -” statements.

#### user\_logic.vhd

```

--- ADD USER PORTS BELOW THIS LINE -----
--- USER ports added here
o_Motor1 : out std_logic_vector(1 downto 0 ) ;
o_Motor2 : out std_logic_vector(1 downto 0 ) ;
--- ADD USER PORTS ABOVE THIS LINE -----

```

#### user\_logic.vhd

```

---USER logic implementation added here
M_CTRL : entity motor_controller
port map (
    i_clk    => Bus2IP_Clk ,
    i_nRst   => Bus2IP_Reset ,
    i_period => slv_reg0( 24 to 31 ) ,
    i_dir    => slv_reg0( 23 ) ,
    i_run    => slv_reg0( 22 ) ,
    o_M1A    => o_Motor1( 1 ) ,
    o_M1B    => o_Motor1( 0 ) ,
    o_M2A    => o_Motor2( 1 ) ,
    o_M2B    => o_Motor2( 0 )
) ;

```

#### gecko3motion.vhd

```

--- ADD USER PORTS BELOW THIS LINE -----
---USER ports added here
Motor1 : out std_logic_vector( 1 downto 0 ) ;
Motor2 : out std_logic_vector( 1 downto 0 ) ;
--- ADD USER PORTS ABOVE THIS LINE -----

```

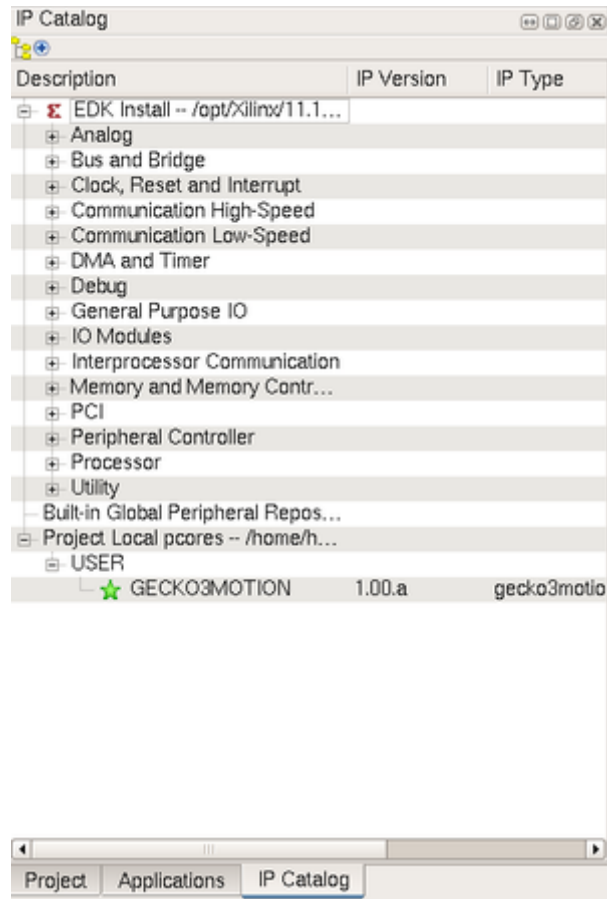
#### gecko3motion.vhd

```

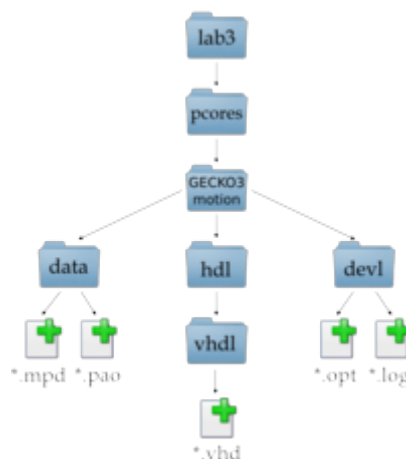
--- MAP USER PORTS BELOW THIS LINE -----
--- USER ports mapped here
o_Motor1 => Motor1 ,
o_Motor2 => Motor2 ,
--- MAP USER PORTS ABOVE THIS LINE -----

```





The peripheral that you just added becomes part of the available core list. Use a file browser to browse to the project directory and ensure that the structure similar to the one shown in the figure below has been created by the Create and Import Peripheral Wizard.



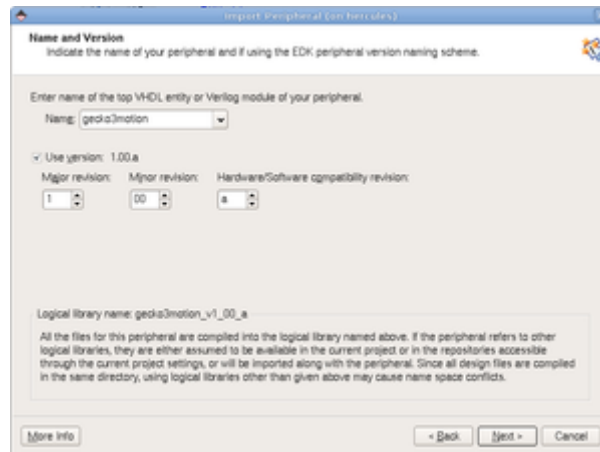
## Import the Gecko3Motion Peripheral

Now, the wizard is going to be used in import mode.

- Select from the menu “Hardware→Create or Import Peripheral” and click “Next”.
- Select “Import existing peripheral” and click “Next”.
- Select “To an XPS project”, ensure that the folder chosen is the project folder, where the files are located. Afterwards, click “Next”.

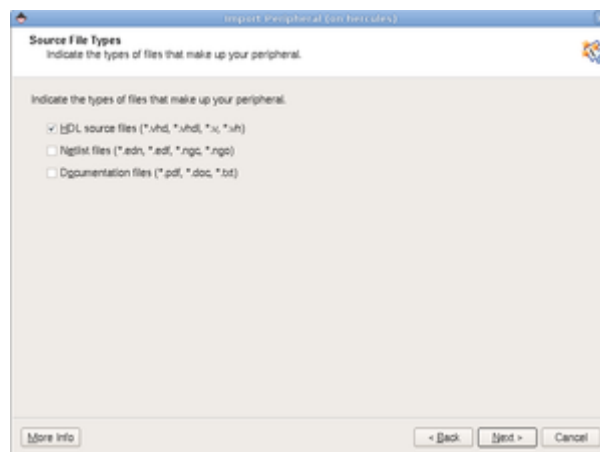
For the name of the peripheral, type “gecko3motion”. Tick “Use version” and select the same version number that originally has been given. Click “Next”. The wizard will ask if we are willing to overwrite the existing

peripheral. Answer this question with “Yes”.



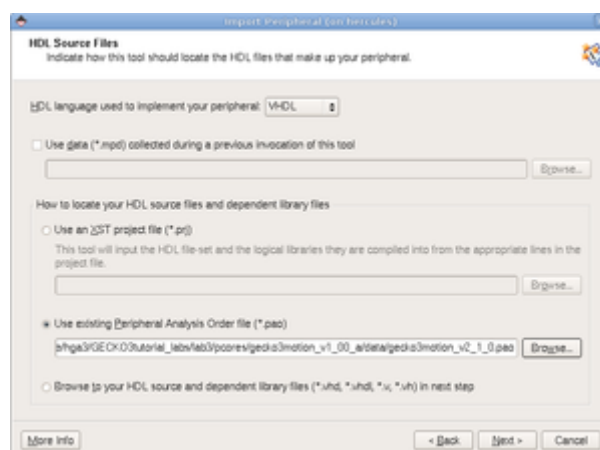
The dialog box is titled "Import Peripheral (not for codes)". It has a tab labeled "Name and Version". The instruction says "Indicate the name of your peripheral and if using the EDK peripheral version naming scheme." There is a text field for "Name" containing "gecko3motion". Below it, there is a checkbox "Use version: 1.00a" which is checked. To the right of this checkbox are three spin boxes for "Major revision" (set to 1), "Minor revision" (set to 00), and "Hardware/Software compatibility revision" (set to a). Below these is a text field for "Logical library name" containing "gecko3motion\_v1\_00\_a". A paragraph of text explains that all files for this peripheral are compiled into the logical library named above. At the bottom are buttons for "More Info", "< Back", "Next >", and "Cancel".

Tick “HDL source files” and click “Next”.



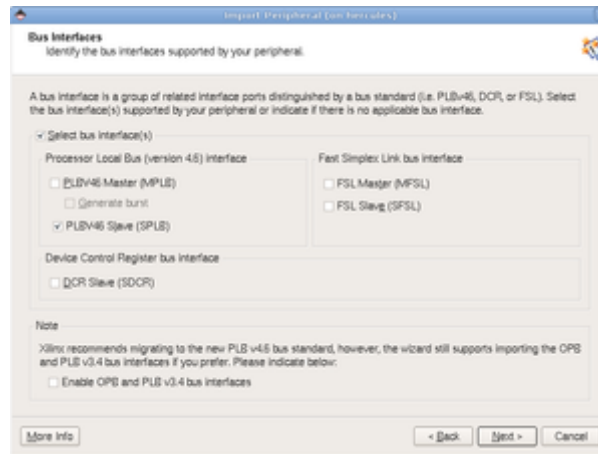
The dialog box is titled "Import Peripheral (not for codes)". It has a tab labeled "Source File Types". The instruction says "Indicate the types of files that make up your peripheral." There are three checkboxes: "HDL source files (\*.vhd, \*.vhdl, \*.v, \*.vh)" which is checked, "Nglint files (\*.edn, \*.edl, \*.ngc, \*.ngc)" which is unchecked, and "Documentation files (\*.pdf, \*.doc, \*.txt)" which is unchecked. At the bottom are buttons for "More Info", "< Back", "Next >", and "Cancel".

Select “Use existing Peripheral Analysis Order file (\*.pao)” and click “Browse”. From the project folder, go to “pcores/gecko3motion\_v1\_00\_a/data” and select the “gecko3motion\_v2\_1\_0.pao” file. Click “Next”.

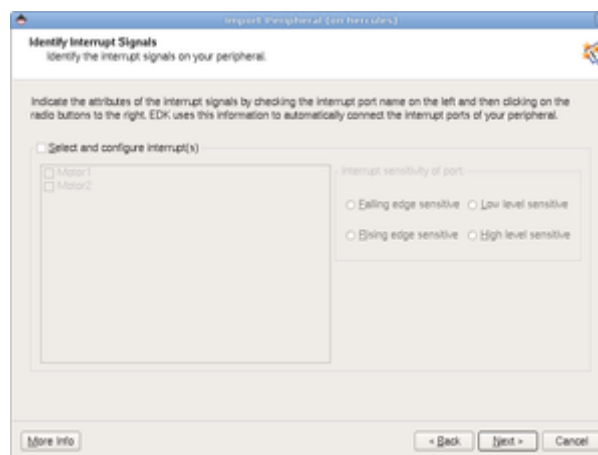


The dialog box is titled "Import Peripheral (not for codes)". It has a tab labeled "HDL Source Files". The instruction says "Indicate how this tool should locate the HDL files that make up your peripheral." There is a dropdown menu for "HDL language used to implement your peripheral" set to "VHDL". Below it is a checkbox "Use data (\*.mpd) collected during a previous invocation of this tool" which is unchecked. There is a text field and a "Browse..." button. Below that is a section "How to locate your HDL source files and dependent library files" with three options: "Use an iST project file (\*.prj)" (unchecked), "Use existing Peripheral Analysis Order file (\*.pao)" (checked), and "Browse to your HDL source and dependent library files (\*.vhd, \*.vhdl, \*.v, \*.vh) in next step" (unchecked). The "Use existing Peripheral Analysis Order file (\*.pao)" option has a text field containing the path "h:\pcores\gecko3motion\_v1\_00\_a\data\gecko3motion\_v2\_1\_0.pao" and a "Browse..." button. At the bottom are buttons for "More Info", "< Back", "Next >", and "Cancel".

On the HDL analysis information page, if you scroll down, you will see the “motor\_controller.vhd” file listed third from the bottom. Click “Next”. The wizard will mention if any errors are found in the design. On the Bus Interfaces page, tick “PLB Slave” and click “Next”.



Disable the interrupt support for the two external ports because there is no point in supporting interrupts over this ports.



Click through the wizard until the button change to finish. At that point read the summary and make sure it is what you wanted. After clicking on finish the core is ready to use. The motor controller peripheral should be accessible through the “IP Catalog→Project Repository”, now.

□ Sketch on a paper what you did. Begin with the bus interface on the top and end up with the motor interface at the bottom (block diagram).

For using the core put an instance into the SoC design and connect it to the system bus (PLB). This is the same process as before (GPIO block connection), by the way. Refresh the address mapping and create the ports (external nets) for controlling the H-bridge. The external ports must be mapped by the UCF description to the right FPGA pins. Listing below gives the necessary help for this step. If every thing has been worked out successfully, the design will be ready for a further step. Build the new hardware (regenerate the netlist) as well as the libraries due to the changes. Afterwards, download the corresponding C source file that has to be adapted in case of different naming. After building the project test if it is working as you expect on hardware. Do not forget to open a serial terminal session for checking the firmware outputs.

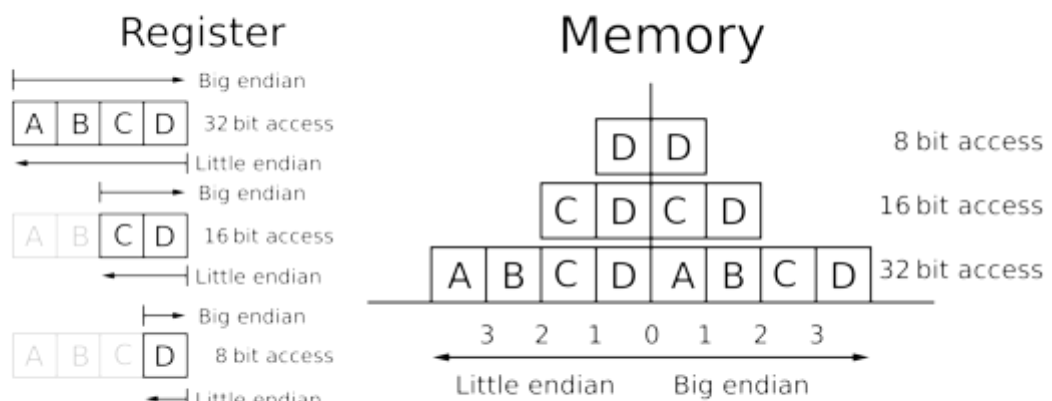
### ucf\_extention

```
# Motors
NET motors_robot_Motor1_pin<0> LOC = AA21;
NET motors_robot_Motor1_pin<1> LOC = AB24;
NET motors_robot_Motor2_pin<0> LOC = AC26;
NET motors_robot_Motor2_pin<1> LOC = W24 ;
```

## Write a test application for the motors

### Write a test application

Make sure you understand the signal mapping. Be aware of the endianness that is different. Get more information about endianness in the figure below.



Now, write a simple software application for accessing the slave register. First of all, copy the code in the listing below. This source shows how to read or write to a slave register. Make sure the header file of the IP is included (get the necessary information by the second listing). The driver file of the IP (stored in the data folder) consists the possibility to define some macros for making registers or bit fields easier accessible.

### test\_proc.c

```
/*
 * Test application motor controller
 */
xil_printf("User logic slave module (motor controller) test...\n\r");
xil_printf(" - write 1 to slave register 0\n\r" );

GECK03MOTION_mWriteSlaveReg0( baseaddrMotion , 0, (Xuint32) 819);
Reg32Value = GECK03MOTION_mReadSlaveReg0( baseaddrMotion, 0);

xil_printf(" - read %d from register 0\n\r" , Reg32Value );
if( Reg32Value != ( Xuint32 ) 819 )
{
    xil_printf(" - slave register 0 write/read failed\n\r" );
    return XST_FAILURE;
}
```

### inc\_ex

```
// include Driver
#include "gecko3motion.h"
```

## Verify the design in hardware

Generate the bit-stream and download it. After completing the download the wheels will run on a default speed.

## Analyse exercises

To get an overview about what is going on, do some core analysis.

- ☐ Analyse the folder pcores
- ☐ Analyse the folder drivers

## Write a software application

To manipulate the speed and the direction, write a software applications, lets call it the firmware. Make sure the firmware communicates over the UART interface. As you know there are three buttons, use two of them for speed up and down controlling (functionally similar to dimming). The last button should be used for changing the direction.

To get some guidelines follow the described steps:

1. Read the buttons in and invert the result due to hardware's pull-up mode.
2. Write a function that increment or decrement the period, respectively, depending on the pushed button.
3. Use the third button to control the direction.
4. Use a switch to simulate an electrical brake (additionally). Get the necessary information from

the wiki.

## IPIF port information

### Bus2IP\_Clk

Synchronization clock provided to the user logic. All IPIC signals are synchronous to this clock. It is identical to the input <bus>\_Clk signal of the peripheral. No additional buffering is provided on the clock; it is passed through as is.

### Bus2IP\_Reset

Active high reset for used by the user logic; it is asserted whenever the <bus>\_Rst signal asserts or whenever there is a software-programmed reset (if the soft reset block is included).

### Bus2IP\_Data

Write data bus to the user logic. Write data is accepted by the user logic during a write operation by assertion of the write acknowledgement signal and the rising edge of the Bus2IP\_Clk.

### Bus2IP\_BE

Byte Enable qualifiers for the requested read or write operation to the user logic. A bit in the Bus2IP\_BE set to '1' indicates that the associated byte lane contains valid data. For example, if Bus2IP\_BE = 0011, this indicates that byte lanes 2 and 3 contains valid data.

**Bus2IP\_RdCE**

Active high chip enable bus to the user logic. These chip enables are asserted only during active read transaction requests with the target address space and in conjunction with the corresponding sub-address within the space. Typically used for user logic readable registers selection.

**Bus2IP\_WrCE**

Active high chip enable bus to the user logic. These chip enables are asserted only during active write transaction requests with the target address space and in conjunction with the corresponding sub-address within the space. Typically used for user logic writable registers selection.

**IP2Bus\_Data**

Output read data bus from the user logic; data is qualified with the assertion of IP2Bus\_RdAck signal and the rising edge of the Bus2IP\_Clk.

**IP2Bus\_RdAck**

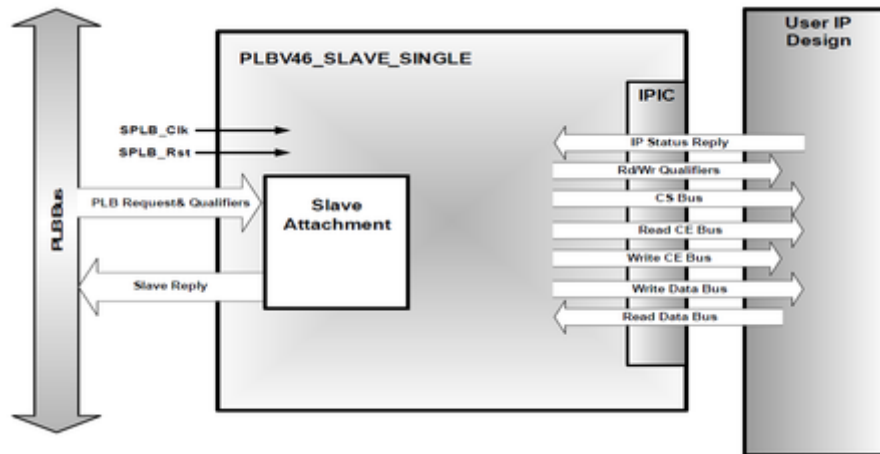
Active high read data qualifier providing the read acknowledgement from the user logic. Read data on the IP2Bus\_Data bus is deemed valid at the rising edge of the Bus2IP\_Clk and IP2Bus\_RdAck asserted high by the user logic. For immediate acknowledgement (such as for a register read), this signal can be tied to '1'. Wait states can be inserted in the transaction by delaying the assertion of the acknowledgement.

**IP2Bus\_WrAck**

Active high write data qualifier providing the write acknowledgement from the user logic. Write data on the Bus2IP\_Data bus is deemed accepted by the user logic at the rising edge of the Bus2IP\_Clk and IP2Bus\_WrAck asserted high by the user logic. For immediate acknowledgement (such as for a register write), this signal can be tied to '1'. Wait states can be inserted in the transaction by delaying the assertion of the acknowledgement.

**IP2Bus\_Error**

Active high signal indicating the user logic has encountered an error with the requested operation. It is asserted in conjunction with the read/write acknowledgement signal(s).



## Conclusion

Use the “Create and Import Peripheral Wizard” to integrate a user specific peripheral into an existing processor system. The wizard creates the necessary directory structure and adds the necessary files (MPD, PAO) to the project directory. After creating a peripheral use it in the design by following the same XPS work-flow that you have learned during previous exercises. Once the peripheral is incorporated in the design, generate the bit-stream and verify its function in hardware. Furthermore, you have learned how to write basic firmware. The firmware is going to be used for controlling the system.

To get used to the complex interfaces, you have learned how to use the wizard and its automatically generated templates. Not only is a template a simplification, but also it helps avoiding errors.

## Solutions

- [Lab3 design as tar archive](#)

# Lab4: Write a Software Application

## Introduction

This lab guides you through the process of writing a basic software application. The software you are going to write, will control the LEDs on the GECKO3main. You will add also a Block-RAM (BRAM) controller, accessible over the PLB, and modify the linker script so that the text section (executable instructions of a program) of the software application will be stored in the Block-RAM (additional information to the linker script are given in the subsection at the end of this exercise). Finally, verify the design on hardware.

## Objectives

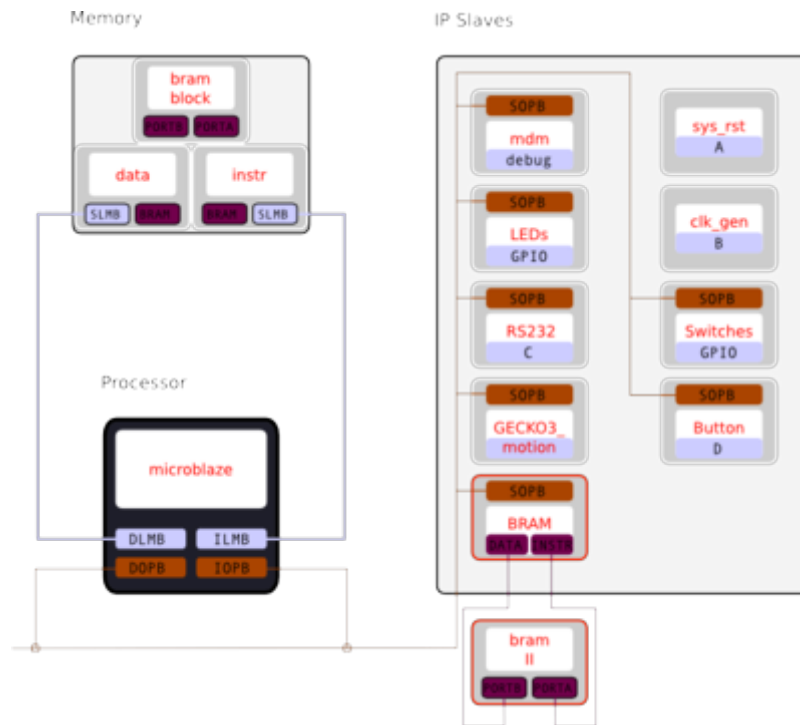
After working out the first lab, you will be able to:

- Add an internal BRAM memory controller
- Write a basic application for accessing the LEDs
- Partition the executable sections onto both the LMB and external memory spaces
- Modify the linker script to realise a partitioning of the firmware over different memory areas
- Generate the bit file
- Download the bit file and verify it on the target (GECKO3EDU)

## Procedure

The first three labs defined the hardware for the soft processor system. This exercise comprises several steps, adding a memory controller and writing a basic software application for accessing the LEDs on the GECKO3EDU. At each general instruction for a given procedure, you will find accompanying step-by-step directives, illustrated by figures providing more details to general instruction. If you feel confident about a specific instruction, feel free to skip the step-by-step directive and move on to the next topic in the procedure. The final design is visualized in the figure below.





## Open the Project

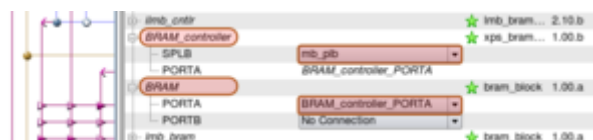
Create a lab4 folder and copy the contents of the lab2 or lab3 folder into the new lab4 folder, if you wish to continue with the design you have created during previous exercises. Check the directory's content, quickly, for verification.

## Add Block RAMs

After starting XPS select “Open Recent Project” to open the project located in the lab4 directory. From the IP catalog, add the following IPs to the embedded hardware system:

- XPS BRAM controller 1.00.b
- Block RAM (BRAM) block 1.00.a

Change the instance names and establish the connections according to the figure below.



In the **addresses** tab, generate the new address mapping after adding and connecting the controller and the BRAM, respectively. Regenerate the address map by the button “Generate Addresses”.

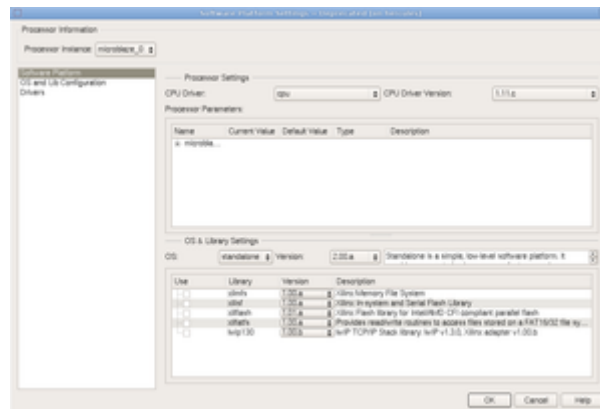
Make sure the controller is not bound to the address 0x00, if so, unbound it and try again.

manually. After all, the “Addresses” tab should look like the one shown in figure below..

Instance	Base Name	Base Address	High Address	Size	Bus Interface	Bus Name	IP Type	IP Version
microblaze_0	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	C_BASERACOR	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

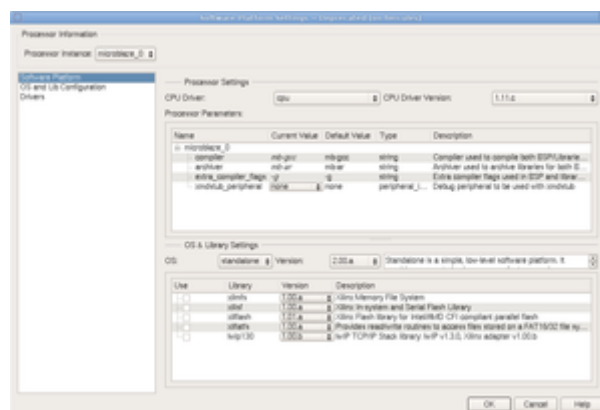
## Create a BSP (Basic Software Project)

Select **Software** → **Software Platform Settings** to change the **MicroBlaze** basic settings (figure below). But first read the release specific notes given by Xilinx (pop-up window). After noticing it confirm by click on the OK button.

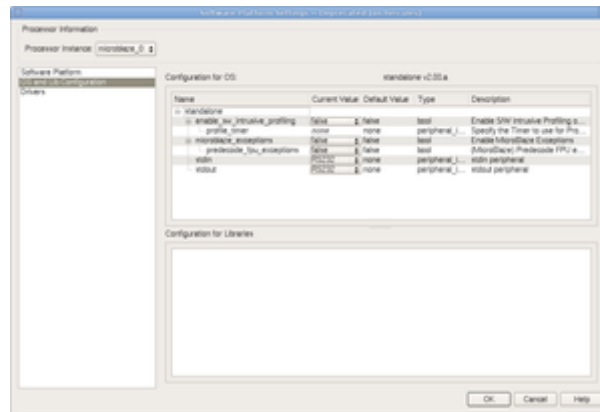


**Note:** You can select the driver for each of the peripherals used in the system. In case of using an operating system, select the appropriate kernel for each IP instance. In addition, select supporting libraries, if needed. In systems where different driver versions are available, provide this option the designer with the possibility of making changes in an easy manner.

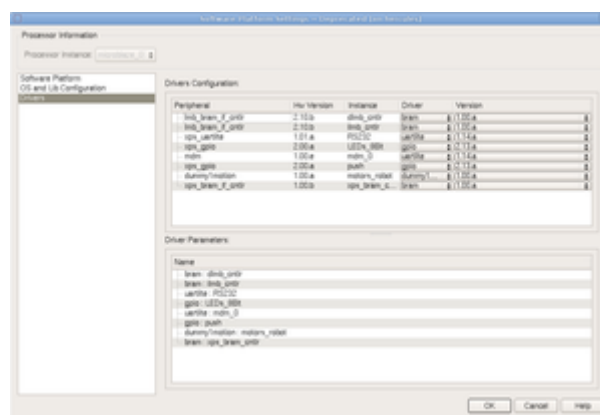
Select the “Software Platform” tab and verify the parameters shown in figure (make changes if necessary):




Afterwards select the “OS and Libraries” tab and verify if the RS232 is selected as stdin as well as stdout (this is required for getting debug information out over the selected channel). Keep the current value for **microblaze\_exceptions** but set the value for **enable\_sw\_intrusive\_profiling** to “false”. Verify, if the settings corresponds to the one shown in figure below.



In the drivers section, review the list of drivers for each peripheral used in the system. The driver files of the user created IPs are located in the directory drivers. If you have more than one driver for a certain IP suitable, select the one that corresponds best to your preferences.



After confirming all changes, click OK to finish this step. XPS writes the parameters that has been specified in the “Software Platform Settings” into a file called **system.mss**. If you like, verify the settings in the **system.mss** file. It is worth doing it for getting a feeling how XPS works.

Generate the BSP by **Software → Generate Libraries and BSPs** or by this  button. This command runs LibGen that relies on the settings in the system.mss file.

**Note:** The suffix mss means Microprocessor Software Specification. In this file is the enclosing description of the used core versions with the corresponding drivers. If you analyse the descriptions of the softcore processor (here MicroBlaze) you find also the name of the used compiler (here mb-gcc), which is going to be used for generating all binary files.

☐ List the created subdirectories , their content, and their possible purposes as a brief description of the microblaze\_0 folder.

## Update Basic C File

During the previous exercise, the topic how to add and access IPs had been introduced. Now it is high time to access the LEDs with the new achieved knowledge. You may remember the design procedure used for accessing the switches and buttons, respectively. The LEDs had been connected by the wizard template during completion of the first laboratory exercise. Check the UCF file shown below as for help routing the LEDs to the correct I/O banks.

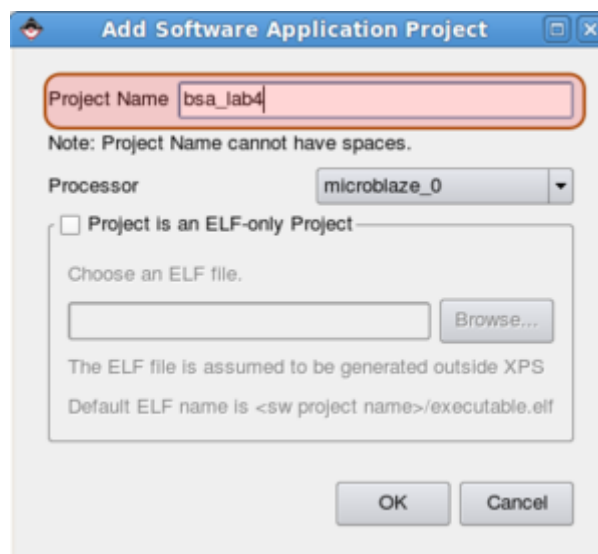
For getting specific information use the gecko3 wiki website. Ones you work on a project based on gecko3 the wiki will provide all necessary information for working with gecko.

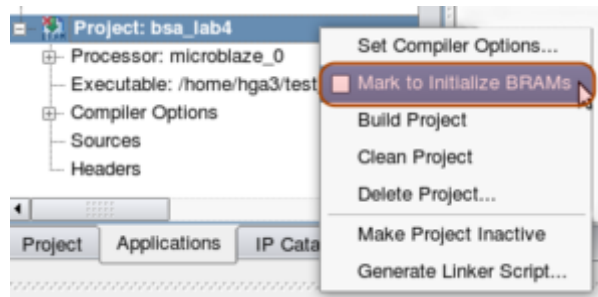
### system\_sniped.ucf

```
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<0> LOC=f9;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<0> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<0> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<1> LOC=g9;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<1> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<1> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<2> LOC=f10;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<2> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<2> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<3> LOC=e10;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<3> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<3> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<4> LOC=c12;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<4> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<4> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<5> LOC=e13;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<5> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<5> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<6> LOC=d13;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<6> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<6> SLEW = SLOW;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<7> LOC=c13;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<7> IOSTANDARD = LVCMOS33;
Net fpga_0_LEDs_8Bit_GPIO_d_out_pin<7> SLEW = SLOW;
```

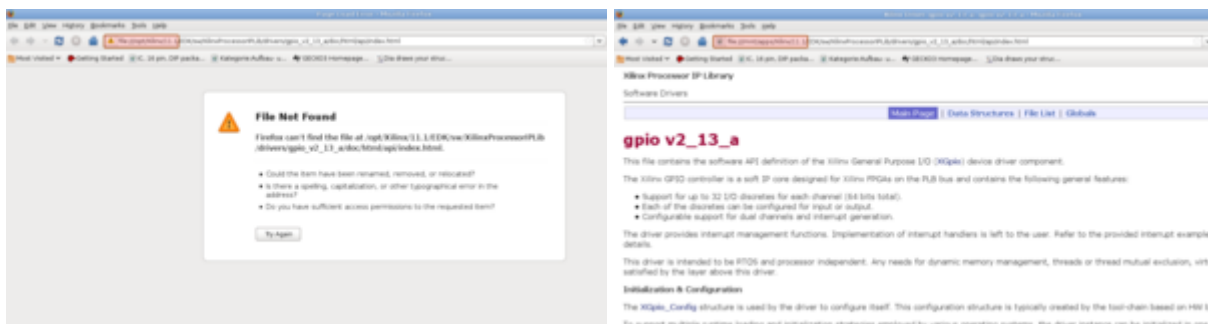
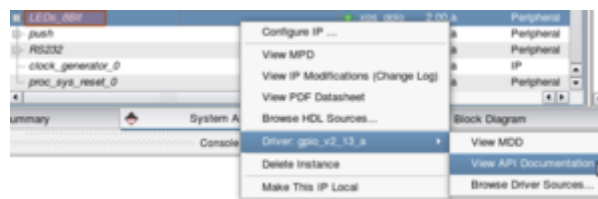
The net name of the LED pins could differ from the one shown above. No-worries, the names must correspond to the HW design and not to a description given. At that point the snippet provides general information concerning the pin location and the operation mode, only.

In the applications tab, add a new SW application. Name the application “bsa\_lab4” as shown in the figure below. Just as a little reminder, the ELF file that is taken is the one marked as active. A project is active if the option “Make to initialise BRAMs” is ticked, as shown below.





Extend the functionality of lab4.c by adding code for displaying the buttons' state or the switches' position, respectively, by the LEDs. To get information about the API, right-click on the LEDs\_8Bit peripheral in the "System Assembly View: gpio\_version → View API Documentation". As shown below. It will start an instance of FireFox. Make sure FireFox is started on the server therefore the **"-no-remote"** option must be set. Have a look in the [intro section](#) for more details.



Browse through the documentation for getting information concerning the core. Read information to the header file **xgpio.h** and review the list of available function calls as introduction to the API.

The following actions have to be performed by the software application to enable write access to the GPIO core:

1. Initialize the GPIO [ `XGpio_Initialize ()` ]
2. Set data direction [ `XGpio_SetDataDirection()` ]
3. Write the data [ `XGpio_DiscreteWrite()` ]

Find the descriptions to the following functions:

#### • **XGpio\_Initialize (XGpio \*InstancePtr , Xuint16 DeviceId)**

**InstancePtr** is a pointer to an Xgpio instance. The memory pointer references must be preallocated by the caller. Further calls to manipulate the component through the XGpio API must be made with this pointer.

**DeviceId** is the unique ID of the device controlled by this XGpio component. Passing in a device ID associates the generic XGpio instance to a specific device, as chosen by the caller or application developer.

#### **XGpio\_SetDataDirection (XGpio \* InstancePtr , unsigned Channel, Xuint32 DirectionMask)**

**InstancePtr** is a pointer to the XGpio instance to be worked on.

**Channel** indicates the channel of the GPIO (1 or 2) to operate on.

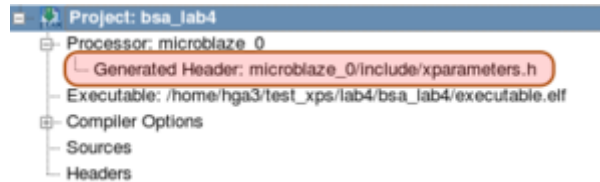
**DirectionMask** is a bit mask specifying bidwise the data direction. Bits set to 0 are declared as outputs and bits set to 1 as inputs, respectively.

**XGpio\_DiscreteWrite (XGpio \*InstancePtr , unsigned channel, Xuint32 data )**

**InstancePtr** and **Channel** have the same meaning as in the previous functions.

**Data** is the data written to the discrete register.

In the microblaze\_0 instance of the bsa\_lab4 project in the applications tab, double-click the Generated Header: microblaze\_0/include/xparameters.h entry.



LibGen writes the xparameters.h file, which provides information for accessing the elements by macros. If you wanna access a specific element, look for its specific name in the xparameters.h file. Do not use its addresses due to a lack of firmware portability.

**Note:** The LEDs\_8BIT matches the instance name assigned in the MHS file. The listing below is a snippet of the skeleton file that you should use the start. After download the file make all necessary changes. Where to store and how to add the source is obvious (**<project>/bsa\_lab4/src**). In case the directory had not been created, create it in a shell.

```
mkdir <NAME>
```

**lab4\_skel.c**

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"
//=====
int main (void)
{
    XGpio push, dip;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");

    // init push buttons
    XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    // init switches
    XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    // init LEDs
    /* Place here your initialization code */

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        dip_check = XGpio_DiscreteRead(&dip, 1);

        xil_printf("\rPush Button State: %x | ", psb_check);
        xil_printf("Dip Switches State: %x", dip_check);

        /* place here your code of the first exercise
        * Ex. 1 :
        *   Write a little SW application that makes the
        *   LEDs blinking.
        */
    }
}
```

```

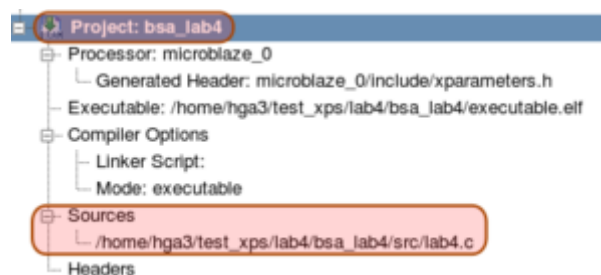
/* place here your code of the second exercise
 * Ex. 2 :
 *   Realise an binary counter displayed by the LEDs.
 */


/* place here your code of the third exercise
 * Ex. 3 :
 *   As for visualisation the buttons state use the
 *   LEDs (red -> button off | green -> button on)
 *   Hint : Use button 1-3 and the corresponding LEDs
 *           next to each button.
 *   Discuss : Think about an other and much easier
 *              solution than the SW one.
 */

    for (i=0; i<999999; i++);          // poor wait cycle ;- )
} // end while(true)
} // end main

```

□ Complete exercise on to three in (a) separate files or (b) in the skeleton file given. The exercise descriptions are in the skeleton file.




After editing, save all changes and click the compile button  as for compiling the program. Verify each exercise on the target hardware, if it works go further to the next exercises until all three exercises has been tested on hardware. The exercises are designed in a way that you can recycle gained information.

## Test changes

As for analyzing the linker script follow the descriptions given in section “understanding the linker script”. Make screen shots of the objdump outputs (more information are given later). Compare and identify differences. Compare and contrast the two situations by a discussion with one of your class mates.

## Understanding the linker script

Analyze the assembled object files with the tool objdump. For using it, start a Cygwin shell by clicking on  or under “Project → Launch Xilinx Bash Shell”. Go to the work directory by using the command

```
cd
```

In the actual work directory is a sub-folder named after the SW project (e.g. bsa\_lab4). (For getting out the actual directory path, referenced to root, use the command pwd.) To get some information about your executable “ELF” (Executable Linked File) use the following command

```
mb-objdump -h <FILE_NAME>
```

When using this command you get as output similar to the information shown in the figure shown below. After modifying the linker script the output will differ. A straightforward method for comparing this two outputs is the screen shot approach as mentioned before. If you like you can work with **diff** instead, only if you know how to use “diff” otherwise work with screen shots.

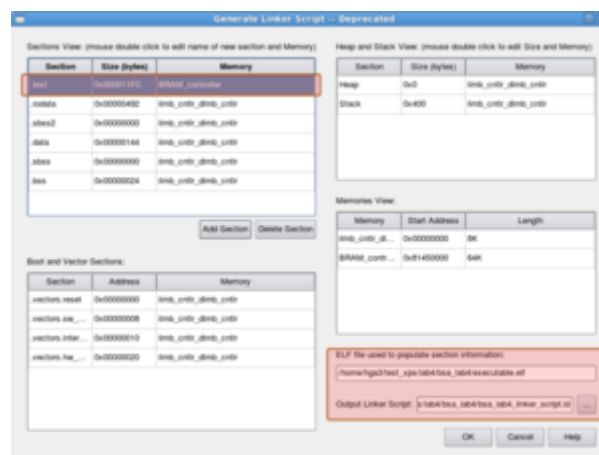
```
bash-4.0$ objdump -h executable.elf
executable.elf:      file format elf32-big

Sections:
Idx Name              Size      VMA           LMA           File off  Align
0 .vectors.reset      00000004  00000000  00000000  000000f4  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .vectors.sw_exception 00000004  00000008  00000008  000000f8  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
2 .vectors.interrupt  00000004  00000010  00000010  000000fc  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
3 .vectors.hw_exception 00000004  00000020  00000020  00000100  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
4 .text               0000115c  00000050  00000050  00000104  2**2
   CONTENTS, ALLOC, LOAD, CODE
5 .init               00000024  000011ac  000011ac  00001260  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
6 .fini               0000001c  000011d0  000011d0  00001284  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
7 .ctors              00000008  000011ec  000011ec  000012a0  2**2
   CONTENTS, ALLOC, LOAD, DATA
8 .dtors              00000008  000011f4  000011f4  000012a8  2**2
   CONTENTS, ALLOC, LOAD, DATA
9 .rodata             00000442  000011fc  000011fc  000012b0  2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
10 .sbss2              00000000  00001640  00001640  00001830  2**0
```

Now change the location of the text section (more information containing this topic is given later. After changing the linker setting file (ld file), recompile the code and re-execute the objdump command. The new and the old (stored output) of objdump should be analysed through comparison.

□ Can you find any changes? (give a short description)

**How to generate a linker script:** Generate your own linker script by right-click on “Project” bsa\_lab4 on the Applications tab and select “Generate Linker Script”. Select BRAM\_controller as the memory location for storing the instructions (.text section! is a part of the software. More information is given in the figure).



Make sure the “**Output Linker Script**” path is set to the bsa\_lab4 directory. Afterwards, click generate that stores the file in the corresponding project directory. After generating, recompile the source for getting a new linked executable, corresponding to the settings made in the ld file. Check it by using again mb-objdump that has to be called in a Cygwin Shell as mentioned before.



```

bash-4.0$ objdump -h executable.elf
executable.elf:      File format elf32-big

Sections:
Idx Name              Size      VMA           LMA           File off  Align
 0 .vectors.reset      00000008  00000000  00000000  000000b4  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .vectors.sw_exception 00000008  00000008  00000008  000000bc  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .vectors.interrupt 00000008  00000010  00000010  000000c4  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .vectors.hw_exception 00000008  00000020  00000020  000000cc  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .text               0000115c  81450000  81450000  00000654  2**2
   CONTENTS, ALLOC, LOAD, CODE
 5 .init               00000024  8145115c  8145115c  000017c0  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 6 .fini               0000001c  81451180  81451180  000017e4  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 7 .rodata             00000442  00000050  00000050  000000d4  2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 8 .sbss               00000000  00000492  00000492  00001800  2**0
   CONTENTS
 9 .data               00000134  00000494  00000494  00000518  2**2
   CONTENTS, ALLOC, LOAD, DATA
10 .ctors              00000008  000005c8  000005c8  0000064c  2**2
   CONTENTS, ALLOC, LOAD, DATA

```

Generate and download the bitstream as for verifying its functionality. Switch the DIP switches as for checking if any action, corresponding to the position, take place (on-off switching of the corresponding LED).

- ☐ Why can a modification of the linker script be helpful?
- ☐ Does it make sense store LUT entries in a RAM? Do you know a better place for storing LUT entries and why?

## Additional linking information

In computer science, a linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable, which is also known as a program.

### Text:

In computing, a code segment, also known as a text segment or simply as text, is a phrase used to refer to a portion of memory or of an object file that contains executable instructions. It has a fixed size and is usually read-only. If the text section is not read-only, then the particular architecture allows self-modifying code. Read-only code is reentrant if it can be executed by more than one process at the same time.

As a memory region, a code segment resides in the lower parts of memory or at its very bottom, in order to prevent heap and stack overflows from overwriting.

### Data:

The Data area contains global variables used by the program that are not initialized to zero. For instance the string defined by `char s [] = "hello world";` in C would exist in the data part.

### BSS:

The BSS segment starts at the end of the data segment and contains all global variables that are initialised to zero. For instance a variable declared `static int i;` would be contained in the BSS segment.

### Stack:

The stack is a LIFO structure, typically located in the higher parts of memory. It usually "grows down" with every register, immediate value or stack frame being added to it. A stack frame consists at minimum of a return address.

### Heap:

The heap area begins at the end of the data segment and grows to higher addresses from there. The Heap area is managed by `malloc`, `realloc`, and `free`, which use the `brk` and `sbrk` system calls to adjust its size. The Heap area is shared by all shared libraries and dynamic load modules in a process. The heap is used for dynamic memory allocation during runtime of a program. It can be seen as a way of distributing ownership of limited memory resources among many pieces of data and code.

## Conclusion

Use Xilinx Platform Studio to define, develop, and integrate the software components of the embedded system. You can define a device driver interface for each of the peripherals and the processor, respectively. XPS creates a MSS file that represents the software side of the processor system. You can then develop and compile peripheral-specific functional software and generate the executable file from the compiled object codes and libraries. If needed, you can also use a linker script to target various segments in various memories. You can edit the linker script to control where to place various code segments on the target memory.

## Solutions

- [Lab4 design as tar archive](#)

# Lab5: Use a Timer and an IRQ Ctrl

## Introduction

This lab guides you through the process of adding timers to an embedded system and writing a software application that utilises the timers. The Software Development Kit (SDK) will be used to create and debug the software application. It is based on the well known IDE for SW developing called Eclipse. In this lab will you get familiar with a obviously separated design flow for hardware and software. You get also any impressions about the importance of working hand in hand, by building up embedded systems, with the HW or SW development team, respectively.

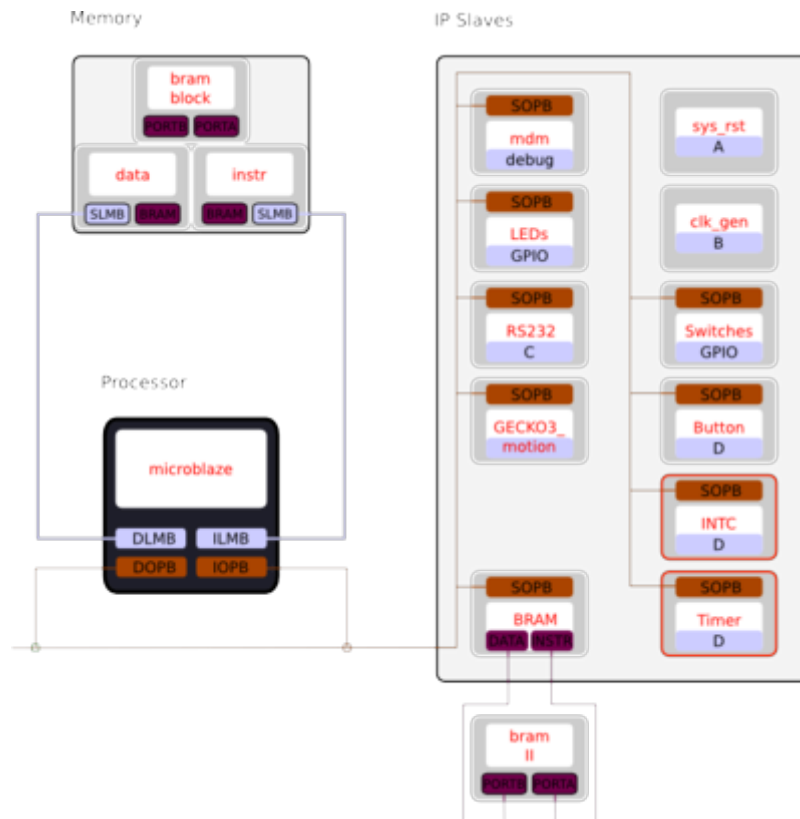
## Objectives

After working out the first lab, you will be able to:

- Utilise a PLB timer with an interrupt controller
- Assign an interrupt handler to the timer
- Instantiate the interrupt handler function in the SW design
- Use the SW IDE as well as the SDK debugger

## Procedure

This lab extends the hardware design carried out in the first set of labs. In this lab you will add a timer and an interrupt controller those will be connected to the SoC by the PLB. After generating the HW design you are going to develop software for using the timer module that generates periodic events (here interrupts for processing time depend actions). By proceeding interrupt mechanism for counting the number of timer events the ability of implementing a system that could be extended for realizing task switching SW will be introduced. (The thoughts are concerned to multitasking.) As for now, we are targeting only simple events that generates some text outputs over the serial interface, not only is this a simple example, it could be easily extended for a lot more complex multitasking system also. This lab comprises several steps, including writing of an interrupt handler used by the software application for accessing the timer. The step-by-step design flow will you ones more cover with a clear and structured design process whereas you get the basic idea out.



## Opening the Project

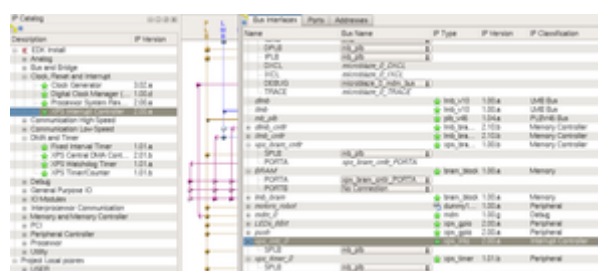
Make a recursive copy (cp -Rf ) of the lab4 folder into the lab5 folder. After copying start XPS over a command line or the starter. By using the tool, open the system.xmp file located in the folder lab5.

```
cp -Rf lab4 lab5
```

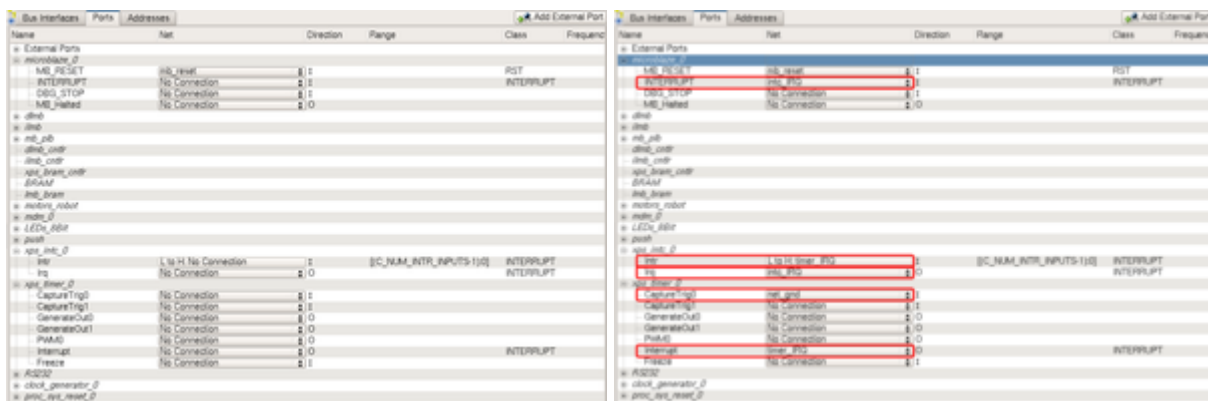
## Add Timer and Interrupt Controller

Add the Timer → **XPS Timer/Counter** (located in the IP catalogue under DMA and Timer) and the interrupt controller → **XPS Interrupt Controller** (located in the IP catalogue under Clock, Reset and Interrupt) peripheral to the design. The instances **xps\_timer\_0** and **xps\_intc\_0** can be added by double-clicking. Those instances are slave devices that can be connected over the PLB. You are free to rename them. It is recommended to rename the timer into the name delay in order to have the ability to follow the descriptions in this tutorial.

Change to the address tab to generate the new address map. First of all click on the button “Generate Addresses”. Afterwards setup the size and let the tool generate the map ones more. In case the resizing had been done without processing the step described before, first. It is more likely you end with a messed-up design than a working one. No-worries in case you did it wrongly. Unbound the components, components are unbounded when the size is set to “U”, and regenerate the address map.



Double-click on the timer to open its parameter setting dialogue box and tick the box “Only One Timer is Present”. Afterwards click OK to accept the changes and close the dialogue box.



Make sure you made the port connections as shown in figure 5.3. Do you know, why using an interrupt controller is important? Is it essential using an interrupt controller for this specific set-up? The answer is definitely no. However, we use an interrupt controller to get the feeling for using it. Here it is simply for education purposes. The timer is used for performing a time depend interrupt. Due to the fact that we have just one interrupt generating core, in this specific HW design, exist the possibility of connecting the timer output (the interrupt output) straight to the INTERRUPT input of the processor (MicroBlaze). However, an advanced HW design would never have just one interrupt generating element. Think about a serial communication that generate interrupts when receiving data. In order to this an interrupt controller comes into the design. This controller covers the MicroBlaze's lack of handling more than only one interrupt. Honestly, it is not really a lack because it is better keeping a design small rather than have a massive core that can handle every thing. The basic ideas are the following:

- The timer generates, depending on the settings, each time an event. Due to the event an interrupt is going to be generated (an IRQ occurs).
- The IRQ is handled by the interrupt controller, which is able to handle more than one interrupt. The interrupts are going to be handled by an hierarchically depending method.
- The controller's IRQ output is connected to the IRQ input of the processor. After receiving

the interrupt the processor dose what every processor would do. Itwill call the corresponding handler, after saving the context, for performing the interrupt.

According to the connection listed below, draw a block diagram including MicroBlaze, PLB and the two new elements. Name the interrupt signals in detail.

Instance Name	Connection
xps_intc	
	Intr → timer_IRQ
	Irq → intc_IRQ
delay	
	CaptureTrig0 → net_gnd
	Interrupt → timer_IRQ
microblaze_0	
	INTERRUPT → intc_IRQ

**Hint:** In the Ports tab, type <PORT\_NAME> for giving the port an user specific name.

- ☐ Why is the port CaptureTrig0 connected to the line net\_gnd?
- ☐ Are there any changes in case of connecting this port to net\_vcc?

After doing it, select **Software → Generate Libraries and BSPs** as for updating the libraries and xparameters.h file, as well.

## Create a C Project (SDK)

First of all export the HW design (Project → Export Hardware Design to SDK). Developing SW with SDK is much more convenient than with XPSs text editor, as you may suppose. What means hardware/software co design. At the begin the HW is going to be designed. In this case hardware is represented by IP based design on a FPGA board, where the board provides interface components. During a second phase firmware is going to be designed and developed for control purposes and communication with general purpose as well as specific IP components.

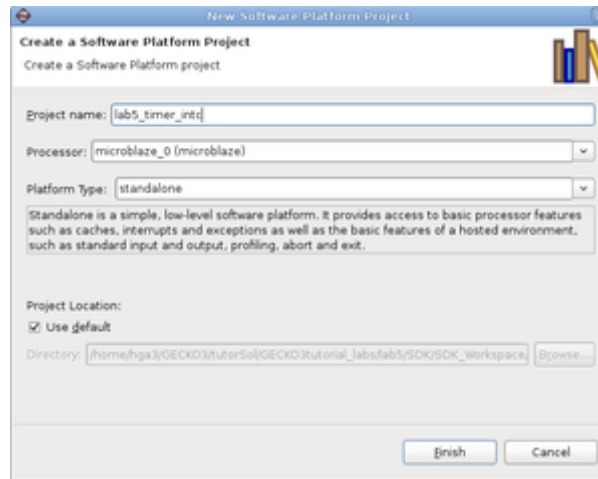
If the SDK starts for the first time, a welcome dialogue appears. It is a good thing to read quickly the information. However, it is not necessary to succeed in doing the tasks described in the exercise. Just in brackets, start SDK locally. Open a new shell and use the given command

```
xps_sdk
```

In addition, there is also a starter in the HuCE-microLab menu.

- **Export the HW Design** - As described above make the design ready, first.
- **Check the environment** - The best way to get an impression about this two tasks is by exploring SDK. However, start the software locally as advised by the shell command or the starter.
- **Choose the env** - Browse to the project directory (workspace) and search for a directory called SDK there is at least one new generated directory (the directory containing the hardware). Make a new folder called "SDK\_workspace" or similar. Set this directory as work directory.
- **Chose the right HW** - After work directory selection, import the HW design. To do so browse to the directory where the HW export is located. There choose a file called **system.xml**, which represents a hardware description based on XML.
- **New SW** - Select **File → New → Software Platform**. Chose a good name for the new SW project (e.g. "lab5\_timer\_intc"). It is a standalone application that runs on the target. Right at the end of the building process the ELF file and syste.bit file will be combined to the so called "download.bit" file.

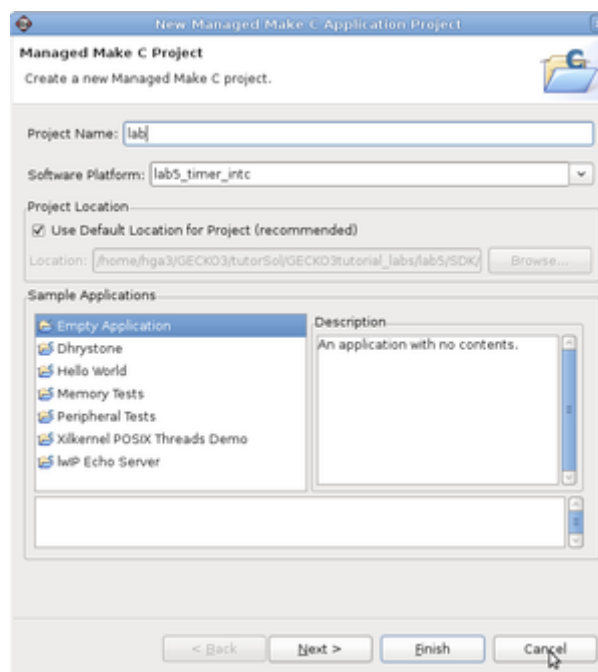
For more detailed information, open the "Cheat Sheets" and read the parts of interest. To get an organized project structure make sub-folder, where source files are kept. Therefore create a C source folder named "src". After walking through the described steps, the design looks like the example shown below.



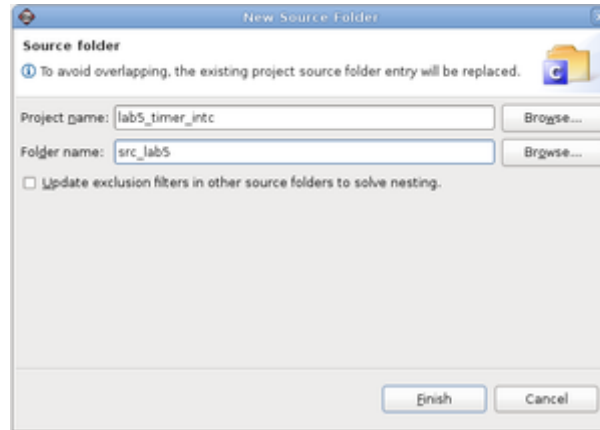
In case noting works, let the IDE create a HW dummy project. Using the resulting ELF file in connection to the HW design as for checking if the problem is based on the hardware or the software part.

## Adding Source Files

Right click on the SW project in the pop-up. There select “New → Managed Make C Application Project”. Chose an empty project and give a good name (e.g. lab5) as shown in the figure.



After having a new C project created and connected to the actual HW design, add a source folder. By the pop-up dialogue. Copy the skeleton files into this source folder. Use whatever you like to do so (file browser, terminal, etc.)



## lab5\_skel.c

```

/*****
 * The LEDs and switches are in these bit positions:
 * LSB 0: gpio_io<3>
 * LSB 1: gpio_io<2>
 * LSB 2: gpio_io<1>
 * LSB 3: gpio_io<0>
 *****/
#include <xmrctr.h>
#include <xintc_l.h>
#include <xgpio.h>
#include <xparameters.h>
#define LEDChan 1

/* Global variables: count is the count displayed using the
 * LEDs, and timer_count is the interrupt frequency.
 */
unsigned int timer_count = 1;
int one_second_flag = 0;

XGpio gpio;

int main() {

    int count_mod_3;
    unsigned int count;
    // Enable MicroBlaze Interrupts
    microblaze_enable_interrupts();

    /* Register the Timer interrupt handler in the vector table */
    XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                        XPAR_XPS_INTC_0_DELAY_INTERRUPT_INTR,
                        (XInterruptHandler) timer_int_handler,
                        (void *)XPAR_DELAY_BASEADDR);

    /* Initialize and set the direction of the GPIO connected to LEDs */
    XGpio_Initialize(&gpio, XPAR_LEDS_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&gpio, LEDChan, 0);

    /* Start the interrupt controller */
    XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);
    XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, 0x1);

    /* Set the gpio as output on high 8 bits (LEDs)*/
    XGpio_mSetDataReg(XPAR_LEDS_8BIT_DEVICE_ID, LEDChan, ~count);
    xil_printf("The value of count = %d\n\r", count);

    /* Set the number of cycles the timer counts before interrupting */
    XTmrCtr_mSetLoadReg(XPAR_DELAY_BASEADDR, 0, (timer_count*timer_count+1) * 80000000);

```



```

/* Reset the timers, and clear interrupts */
XTmrCtr_mSetControlStatusReg(XPAR_DELAY_BASEADDR, 0, XTC_CSR_INT_OCCURED_MASK |
                             XTC_CSR_LOAD_MASK );

/* Enable timer interrupts in the interrupt controller */
XIntc_mEnableIntr(XPAR_DELAY_BASEADDR, XPAR_DELAY_INTERRUPT_MASK);

/* Start the timers */
XTmrCtr_mSetControlStatusReg(XPAR_DELAY_BASEADDR, 0, XTC_CSR_ENABLE_TMR_MASK |
                             XTC_CSR_ENABLE_INT_MASK | XTC_CSR_AUTO_RELOAD_MASK |
                             XTC_CSR_DOWN_COUNT_MASK);

/* Enable MB interrupts */
//microblaze_enable_interrupts();

/* Wait for interrupts to occur */
while(1) {
    if(){
        count_mod_3 = count % 3;
        if(count_mod_3 == 0)
            printf("Interrupt taken at %d seconds \n\r",count);
        //      xil_printf("Interrupt taken at %d seconds \n\r",count);
        one_second_flag=0;
        xil_printf(".");
    }
}
}

```

In the left hand navigator tab, double click on the lab5.c file to open it in an editor. The project is going to be compiled as soon as it will be open. Note report of several compilation errors. Also note the project outline on the right side. There are used libraries and functions listed. This list provides a quick information summary.

## Correct all errors

□ The file lab5\_skel.c contains some errors. Correct those errors and it will compile successfully. Furthermore, you must document the error corrections. If you are not able to find errors talk to class mates or the lecturer.

## Writing the interrupt handler

### Create the interrupt handler

Move on to the part, where the interrupt handler function has been stubbed out - in the source file. Notice the interrupt handler function is called **timer\_int\_handler**. This name must match the name specified in the function call, where the interrupt controller (functions pointer to the call-back procedure) is initialized. If the name does not match exactly, the interrupt handler will not be able to interact properly and therefore it will fail.

### Create a local variables to keep the timer\_int\_handler function's state

#### unsigned int csr ;

The first step when creating a XPS timer interrupt handler, is verifying if the XPS timer generates the supposed interrupts. How to do so is described in the API of the XPS timer controller.

In the “XPS System Assembly View” window, right-click the xps\_timer instance - named delay - and select View PDF Datasheet. Move on to the register description in the data-sheet and study the **TCSR0** register description.

### Timer0 Interrupt

Indicates the condition that an interrupt on the timer has occurred. For reset the indicator bit write a logic one to the register position.

The level 0 driver for the XPS timer provides two functions that read and write from/to the control status register. View the timer API doc by right-clicking on the delay instance - in the system assembly view. Explore by the documentation the following function:

**XTmrCtr\_mGetControlStatusReg(BaseAddress, TmrCtrNumber)** : Returns the control status register state of a timer counter

#### Note:

Substitute *baseaddr* with the base address of the peripheral “delay”. Refer to xparameters.h. To determine if an interrupt occurred (mask the 32 bit value - logic AND - by the XTC\_CSR\_INT\_OCCURED\_MASK parameter). If this is true increment a counter value. Display the counter value by the LEDs\_8Bit peripheral (hint: you may use the XGpio\_mSetDataReg() function). Send the counter value over the RS232 bus to a connected serial-terminal on the host PC (xil\_printf ())

**XTmrCtr\_mSetControlStatusReg(baseaddr, 0, csr );** : This functions resets the interrupt occurred.

[irct\\_ex](#)

```
void timer_int_handler(void * baseaddr_p) {
    /* Add variable declarations here */
    unsigned int csr;
    /* Read timer 0 CSR to see if it raised the interrupt */
    csr = XTmrCtr_mGetControlStatusReg(XPAR_DELAY_BASEADDR, 0);
    /* If the interrupt occurred, then increment a counter */
    if (csr & XTC_CSR_INT_OCCURED_MASK)
        /* Display the count on the LEDs and print it using the UART */
        count++;
    one_second_flag = 1;
    // Display the count on the LEDs and print some statement
    //XTmrCtr_mWriteReg(XPAR_LEDS_8BIT_BASEADDR, 0, 0, count);
    XGpio_DiscreteWrite(&gpioLED, LEDChan, count);
    xil_printf("Count value is: %x\n\r", count);
    /* Clear the timer interrupt */
    XTmrCtr_mSetControlStatusReg(XPAR_DELAY_BASEADDR, 0, csr);
}
```

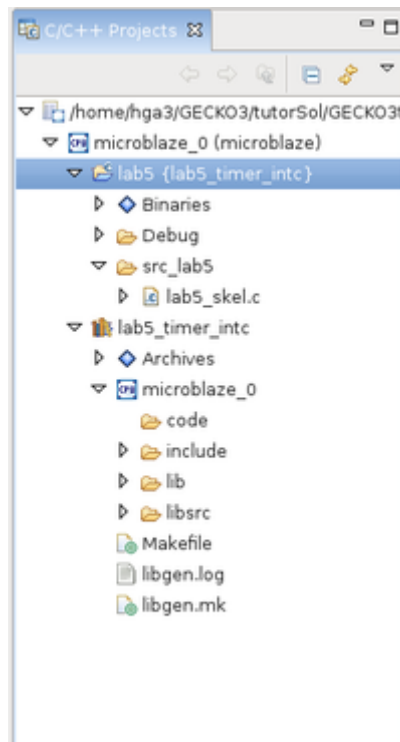
### Analyze compilation output

- ☐ .text segment
- ☐ .data segment
- ☐ .bss segment
- ☐ Total program size in dec
- ☐ Total program size in hex

## Explore the STD C library

- ☐ Use for communication only `**xil_printf()`
- ☐ What is the size of the binary
- ☐ Use for communication only `**printf()` (shipped by the STD C library)
- ☐ Is there a problem? Fix it by a linking modification

On the left hand side of the IDE is the C/C++ tab. There is a folder named lab5 as shown in the figure. After “right click” on the folder, click on the option “properties” in the pop-up menu.



Generate a specific linker script. Check where the **.text** section is kept in the memory. Move it to the BRAM address space.

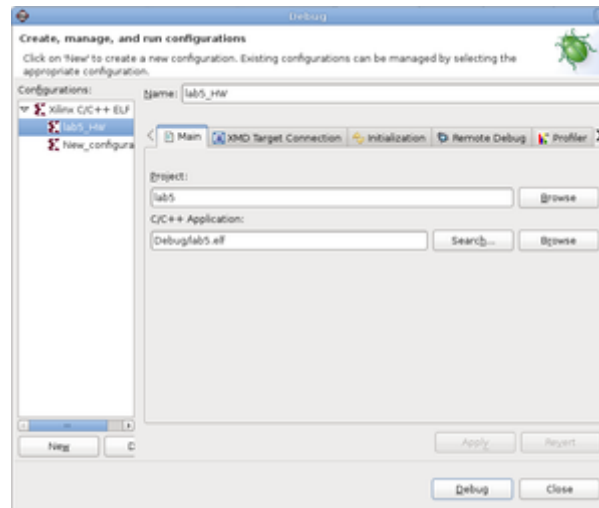
- ☐ Explore the changes made by the linking script modification

Use **objdump**

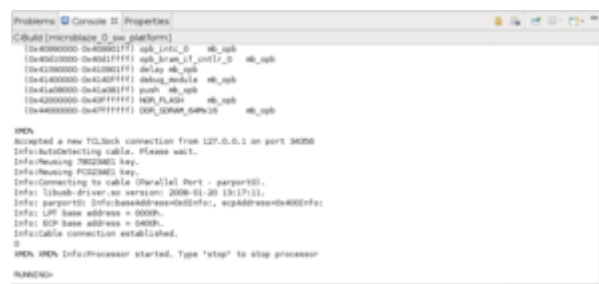
Verify the systems functionality on the GECKO3EDU hardware.

## Debugging using SDK

On the SDK menu, select “Run → Run...”, this will present a screen summarizing the existing configurations. Under configurations, select “Xilinx C/C++ ELF”. Afterwards click on [New] and add the appropriate ELF file. Get inspired by the figure shown below.



Click apply and then the run button for establishing a connection between the debugger and target (e.g. the robot). The output data stream is going to be sent over the RS232 connection whereas the debugging control signals are send over JTAG. Open the "Console View" and click on the terminate button (red button) to terminate the selected running process.



On the SDK menu, select “Run → Debug...”. It will present a screen summarizing the existing configuration. After that click on “Debug”. Confirm the IDE view switch to debug mode. It will open the Debug perspective. The debugger is automatically connected to the processor via XMD. The processor will be interrupted automatically, due to the breakpoint at the first statement in the function `main()`.

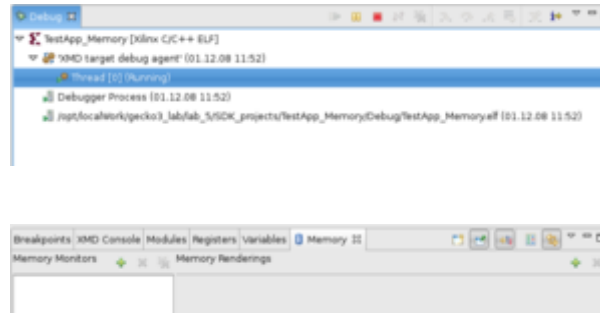
Click on the resume button. If there is an other breakpoint the program will stop there for further instructions or whatever.



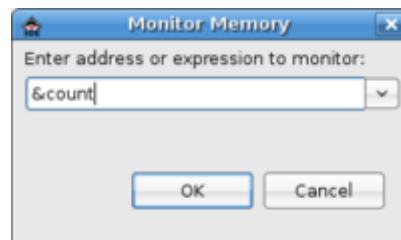
Click on the “Suspend” button. Afterwards “Right click” in the variables tab and choose “Add Global Variables”. All global variables will be displayed. Select count and click [OK]. “Right click” on count and choose “Enable”. Look in the figures below for more information.

Add on line 65 **XGpio\_DiscreteWrite(&gpioLED, LEDChan, count);** in the source code, a breakpoint. Start debugging and observing the count variable on each run through the main loop.

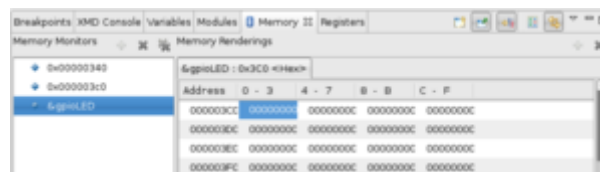




Use the memory monitor. There enter the address of the variable count.



Monitor an other variable e.g. gpioLED. Monitor also the base addresse of LED GPIO function handle. Analyze by the monitor register values. Where are the function pointers to e.g. a interrupt callback function.



Enough played with the debugger? Then terminate the debug session and move on to the next exercise or a first project.

## Conclusion

This lab guided you through the process of adding an XPS timer and interrupt controller, and assigning an interrupt handler function to the interrupting device via the software platform settings. You developed an interrupt handler function and tested it in hardware. Additionally, you used the SDK debugger as for getting an idea of how convenient SoPC developing can be.

## Solutions

- [Lab5 design as tar archive](#)