Writing Basic Software Application

Introduction

This lab guides you through the process of writing a basic software application. The software you will develop will write to the LEDs on the ZedBoard. An AXI BRAM controller with associated 8KB BRAM were added in the last lab. The application will be run from the BRAM by modifying the linker script for the project to place the text section of the application in the BRAM. You will verify that the design operates as expected, by testing in hardware.

Objectives

After completing this lab, you will be able to:

- Write a basic application to access an IP peripheral in SDK
- Develop a linker script
- Partition the executable sections into both the DDR3 and BRAM spaces
- Generate an elf executable file
- Download the bitstream and application and verify on the ZedBoard

Procedure

This lab is separated into steps that consist of general overview statements that provide information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

This lab comprises 4 primary steps: You will open the Vivado project, export to and invoke SDK, create a software project, analyze assembled object files and verify the design in hardware.

Design Description

The design was extended at the end of the previous lab to include a memory controller (see **Figure 1**), and the bitstream should now be available. A basic software application will be developed to access the LEDs on the ZedBoard.

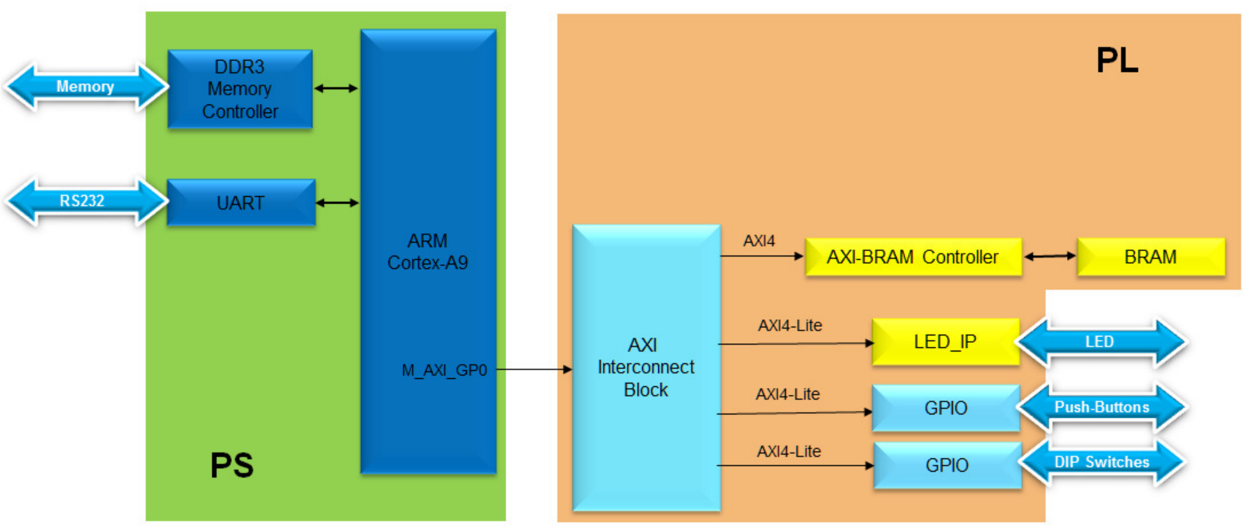


Figure 1 Design used from the Previous Lab

General Flow for this Lab

Step 1: Open the Project	⇨	Step 2: Export to SDK	⇨	Step 3: Analyze Assembled Object File	⇨	Step 4: Verify the design in hardware

Opening the Project Step 1

1-1. Use the lab3 project from the last lab, or use the *lab3* project in *c:\xup\embedded\labsolution,* and save it as *lab4*

1-1-1. Start the Vivado if necessary and open either the lab3 project (lab3.xpr) you created in the previous lab or the lab3 project in the labsolution directory using the **Open Project** link in the Getting Started page.

1-1-2. Select **File > Save Project As …** to open the *Save Project As* dialog box. Enter **lab4** as the project name. Make sure that the *Create Project Subdirectory* option is checked, the project directory path is c:\xup\embedded\labs\ and click **OK**.
This will create the lab4 directory and save the project and associated directory with lab4 name.

Export to SDK and create Application Project Step 2

2-1. Export the hardware along with the generated bitstream to SDK .

To Export the hardware, the block diagram must be open and the Implemented design must be open.

2-1-1. If it is not already open, click **Open Block Design** (under IP Integrator in the Flow Navigator)

2-1-2. Click **Open Implemented Design** (under Implementation)

2-1-3. Start *SDK* by clicking **File > Export > Export Hardware for SDK**...

The export GUI will be displayed.

2-1-4. Check all three checkboxes, including the **Launch SDK** box and click **OK**.

2-1-5. If prompted, click **YES** to overwrite the platform created by the previous lab

2-2. Create an empty project called lab4 using standalone_bsp software platform project. Import lab4.c file from the c:\xup\embedded\sources directory

The hw_platform and bsp projects from the previous lab will automatically rebuild to include the led_ip. Verify this by checking the *system.mss* for the **led_ip** peripheral.

2-2-1. To tidy up the workspace and save unnecessary building of a project that is not being used, right click on the **TestApp** project from the previous lab, and click **Close Project**, as this project will not be used in this lab. It can be reopened later if needed.

2-2-2. Select **File** > **New** > **Application Project.**

2-2-3. Enter **lab4** as the *Project Name*, and for *Board Support Package*, choose **Use Existing** (*standalone_bsp* should be the only option)

2-2-4. Click **Next**, and select *Empty Application* and click **Finish**

2-2-5. Expand **lab4** in the project view, and right-click in the *src folder* and select **Import.**

2-2-6. Expand **General** category and double-click on **File System.**

2-2-7. Browse to **c:\xup\embedded\sources\lab4** folder and click OK

2-2-8. Select **lab4.c** and click **Finish** to add the file to the project. (Ignore any errors for now)

2-2-9. Select the **system.mss** tab.

2-2-10. Click on **Documentation** link corresponding to **btns_5bit** peripheral under the Peripheral Drivers section to open the documentation in a default browser window. As our led_ip is very similar to GPIO, we look at the mentioned documentation.

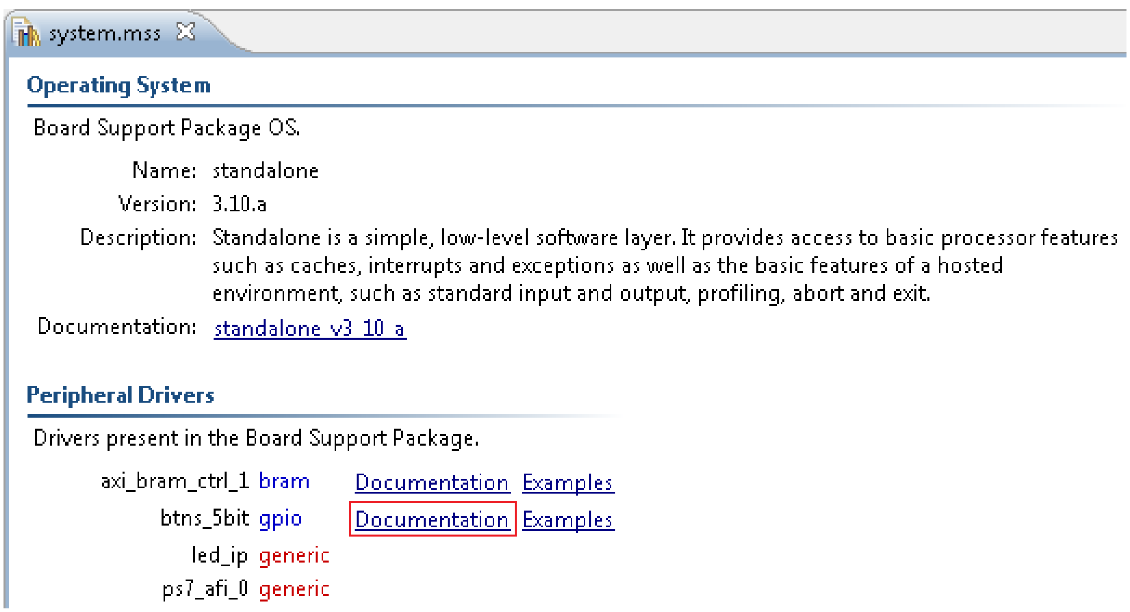


Figure 2 Accessing device driver documentation

2-2-11. View the various C and Header files associated with the GPIO by clicking **Files** at the top of the page.

2-2-12. Click the header file **xgpio.h** and review the list of available function calls for the GPIO.

The following steps must be performed in your software application to enable reading from the GPIO: **1) Initialize the GPIO, 2) Set data direction, and 3) Read the data**

Find the descriptions for the following functions by clicking links:

XGpio_Initialize (XGpio *InstancePtr, u16 DeviceId)

InstancePtr is a pointer to an XGpio instance. The memory the pointer references must be pre-allocated 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, u32 DirectionMask)

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

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

DirectionMask is a bitmask specifying which bits are inputs and which are outputs. Bits set to 0 are output and bits set to 1 are input.

XGpio_DiscreteRead(XGpio *InstancePtr, unsigned channel)

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

Channel *contain*s the channel of the GPIO (1 or 2) to operate on

2-2-13. Double-click on **lab4.c** in the Project Explorer view to open the file. This will populate the **Outline** tab. Open the header file **xparameters.h** by double-clicking on **xparameters.h** in the **Outline** tab

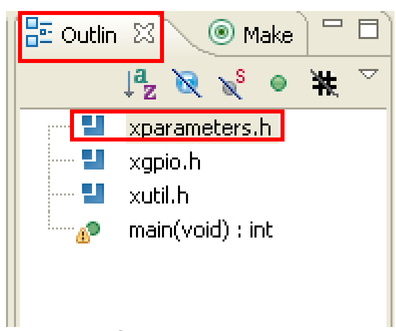


Figure 3 Double-Click the generated header file

The xparameters.h file contains the address map for peripherals in the system. This file is generated from the hardware platform description from Vivado. Find the following #define used to identify the **dip** peripheral:

#define XPAR_SW_8BIT_DEVICE_ID 1 **Note: The number might be different**

Notice the other #define XPAR_SW_8BIT* statements in this section for the 8 bit SW peripheral, and in particular the address of the peripheral defined by: `XPAR_SW_8BIT_BASEADDR`

2-2-14. Modify line 15 of lab4.c to use this macro (#define) in the *XGpio_Intialize* function.

www.xilinx.com/university
xup@xilinx.com

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"

//====================================================

int main (void)
{

   XGpio dip, push;
   int i, psb_check, dip_check;

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

   XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&dip, 1, 0xffffffff);

   XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Modify this
   XGpio_SetDataDirection(&push, 1, 0xffffffff);

   while (1)
   {
     psb_check = XGpio_DiscreteRead(&push, 1);
     xil_printf("Push Buttons Status %x\r\n", psb_check);
     dip_check = XGpio_DiscreteRead(&dip, 1);
     xil_printf("DIP Switch Status %x\r\n", dip_check);

     // output dip switches value on LED_ip device

     for (i=0; i<9999999; i++);
   }
}
```

Figure 4 Imported source, highlighting the code to initialize the SW_8BIT as input, and read from it

2-2-15. Do the same for to the BTNS_5BIT; find the macro (#define) for the BTNS_5BIT peripheral in xparameters.h, and modify line 18 in lab4.c, and save the file.

The project will be rebuilt. If there are any errors, check and fix your code. Your C code will eventually read the value of the switches and output it to the **led_ip.**

2-3. Assign the led_ip driver from the *driver* directory to the led_ip instance.

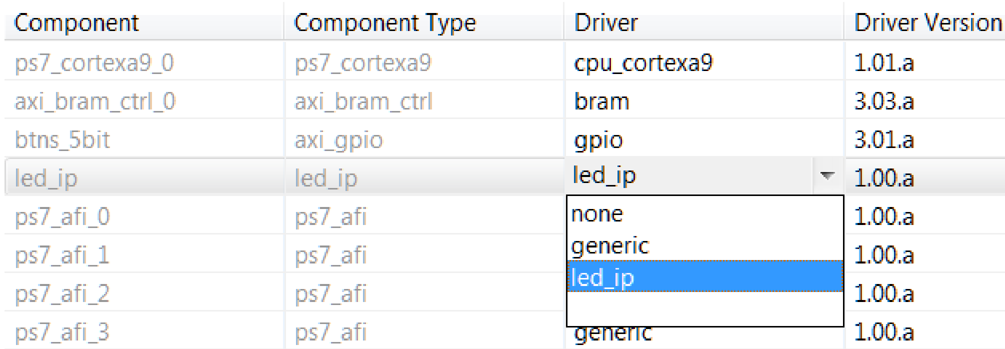
2-3-1. Select **Xilinx Tools** > **Repositories.**

2-3-2. Click on **New** button of *Local Repositories*, browse to *C:\xup\embedded\labs\led_ip\led_ip_1.0* and click **OK**, and click **OK** again to close the Preferences window

2-3-3. Select **standalone_bsp** in the project view, right-click, and select **Board Support Package Settings.**

2-3-4. Select *drivers* on the left (under *Overview*)

2-3-5. Select *Generic* under the *Driver* column for *led_ip* to access the dropdown menu. From the dropdown menu, select **led_ip**, and click **OK.**



Component	Component Type	Driver	Driver Version
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	1.01.a
axi_bram_ctrl_0	axi_bram_ctrl	bram	3.03.a
btns_5bit	axi_gpio	gpio	3.01.a
led_ip	led_ip	led_ip	1.00.a
ps7_afi_0	ps7_afi		1.00.a
ps7_afi_1	ps7_afi		1.00.a
ps7_afi_2	ps7_afi		1.00.a
ps7_afi_3	ps7_afi	generic	1.00.a

Figure 5 Assign led_ip driver

2-4. Examine the Driver code

The driver code was generated automatically when the IP template was created. The driver includes higher level functions which can be called from the user application. The driver will implement the low level functionality used to control your peripheral.

2-4-1. In windows explorer, browse to *C:\xup\embedded\labs\led_ip\led_ip_1.0*\drivers\led_ip_v1_00_a\src. Notice the files in this directory and open led_ip.c. This file only includes the header file for the IP.

2-4-2. Close led_ip.c and open the header file led_ip.h and notice the macros:
LED_IP_mWriteReg(…)
LED_IP_mReadReg(…)

e.g: search for the macro name LED_IP_mWriteReg:

```
/**
 *
 * Write a value to a LED_IP register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param   BaseAddress is the base address of the LED_IP device.
 * @param   RegOffset is the register offset from the base to write to.
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *    void LED_IP_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset,
Xuint32 Data)
 *
 */
#define LED_IP_mWriteReg(BaseAddress, RegOffset, Data) \
Xil_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data))
```

For this driver, you can see the macros are aliases to the lower level functions Xil_Out32( ) and Xil_Out32( ). The macros in this file make up the higher level API of the led_ip driver. If you are writing your own driver for your own IP, you will need to use low level functions like these to read and write from your IP as required. The low level hardware access functions are wrapped in your driver making it easier to use your IP in an Application project.

2-4-3. Modify your C code (see figure below, or you can find modified code in **lab4_sol.c** from **sources** folder) to echo the dip switch settings on the LEDs by using the led_ip driver API macros, and save the application.

2-4-4. Include the header file:

```
#include "led_ip.h"
```

2-4-5. Include the function to write to the IP (insert before the *for* loop):

```
LED_IP_mWriteReg(XPAR_LED_IP_BASEADDR, 0, dip_check);
```

Remember that the hardware address for a peripheral (e.g. the macro **XAR_LED_IP_BASEADDR** in the line above) can be found in *xparameters.h*

```
#include "xparameters.h"
#include "xgpio.h"
#include "xutil.h"
#include "led_ip.h"

//====================================================

int main (void)
{

   XGpio dip, push;
   int i, psb_check, dip_check;

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

   XGpio_Initialize(&dip, XPAR_SW_8BIT_DEVICE_ID);
   XGpio_SetDataDirection(&dip, 1, 0xffffffff);

   XGpio_Initialize(&push, XPAR_BTNS_5BIT_DEVICE_ID);
   XGpio_SetDataDirection(&push, 1, 0xffffffff);


   while (1)
   {
      psb_check = XGpio_DiscreteRead(&push, 1);
      xil_printf("Push Buttons Status %x\r\n", psb_check);
      dip_check = XGpio_DiscreteRead(&dip, 1);
      xil_printf("Switch Status %x\r\n", dip_check);

      // output dip switches value on LED_ip device
      LED_IP_mWriteReg(XPAR_LED_IP_BASEADDR, 0, dip_check);

      for (i=0; i<9999999; i++);
   }
}
```

Figure 6 The completed C file

2-4-6. Save the file and the program will be compiled again.

Analyze Assembled Object Files Step 3

3-1. Launch Shell and objdump lab4.elf and look at the sections it has created.

3-1-1. Launch the shell from SDK by selecting **Xilinx Tools > Launch Shell.**

3-1-2. Change the directory to ***lab4\Debug*** using the **cd** command in the shell.

You can determine your directory path and the current directory contents by using the **pwd** and **dir** commands.

3-1-3. Type ***arm-xilinx-eabi-objdump –h lab4.elf*** at the prompt in the shell window to list various sections of the program, along with the starting address and size of each section

You should see results similar to that below:

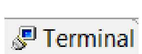
```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001b60  00100000  00100000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         00000018  00101b60  00101b60  00009b60  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         00000018  00101b78  00101b78  00009b78  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       0000018c  00101b90  00101b90  00009b90  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000464  00101d20  00101d20  00009d20  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  00102184  00102184  0000a184  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .mmu_tbl      00004000  00104000  00104000  0000c000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .init_array   00000008  00108000  00108000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .fini_array   00000004  00108008  00108008  00010008  2**2
```

Figure 7 Object dump results - .text in the DDR3 space

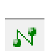
Verify in Hardware Step 4

4-1. Connect and power up the board. Set the Terminal tab to communicate at 115200 baud using correct COM port. Program the FPGA, and run the lab4.elf application.

4-1-1. Connect and power up the board.

4-1-2. Select the 

tab. If it is not visible then select **Window > Show view > Terminal**.

4-1-3. Click on  and select appropriate COM port (depends on your computer), and configure it with 115200 baud rate if necessary

xup@xilinx.com

4-1-4. In SDK, select **Xilinx Tools** > **Program FPGA.**

4-1-5. Click the **Program** button to program the FPGA.

4-1-6. Select **lab4** in *Project Explorer*, right-click and select **Run As** > **Launch on Hardware** to download the application, execute ps7_init, and execute lab4.elf

Flip the DIP switches and verify that the LEDs light according to the switch settings. Verify that you see the results of the DIP switch and Push button settings in SDK Terminal.

```
Serial: (COM11, 115200, 8, 1, None, None - CONNECTED)
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
Push Buttons Status 0
DIP Switch Status D2
```

Figure 8 DIP switch and Push button settings displayed in SDK terminal

Note: Setting the DIP switches and push buttons will change the results displayed.

4-2. Change the first xil_printf function calls to printf. Re-compile the code and observe that the BRAM space is not sufficient. Generate the linker script to target the .text section to external memory (ps7_ddr_0_S_AXI_BASEADDR) and heap and stack sections to the axi_bram_ctrl_0_S_AXI_BASEADDR memory.

4-2-1. In the text editor, in lab4.c, change one of the **xil_printf** function calls to **printf** and save the file.

4-2-2. The code compilation will begin and observe the output in the console window.

```
'Building target: lab4.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T -Wl,../src/lscript.ld
-L../../standalone_bsp/ps7_cortexa9_0/lib -o "lab4.elf"  ./lab4.o
-Wl,--start-group,-lxil,-lgcc,-lc,--end-group
c:/xilinx/sdk/2013.3/gnu/arm/nt/bin/../lib/gcc/arm-xilinx-eabi/4.7.3/../../../.
./arm-xilinx-eabi/bin/ld.exe: lab4.elf section `.text' will not fit in region
`axi_bram_ctrl_0_S_AXI_BASEADDR'
c:/xilinx/sdk/2013.3/gnu/arm/nt/bin/../lib/gcc/arm-xilinx-eabi/4.7.3/../../../.
./arm-xilinx-eabi/bin/ld.exe: region `axi_bram_ctrl_0_S_AXI_BASEADDR'
overflowed by 11264 bytes
collect2.exe: error: ld returned 1 exit status
make: *** [lab4.elf] Error 1
```

Figure 9 Errors Shown in Console Window

The .text section is too big for the allocated section.

4-2-3. Select **Xilinx Tools** > **Generate Linker Script…**

4-2-4. In the *Advanced Tab* change the *.text* section to **ps7_ddr_0_S_AXI_BASEADDR** memory, click **Generate**, and click **Yes** to overwrite.

4-3. Execute the lab4.elf application and observe the application working even when various sections are in different memory.

4-3-1. Select **lab4** in *Project Explorer*, right-click and select **Run As > Launch on Hardware** to download the application, execute ps7_init, and execute lab4.elf

Click *Yes* if prompted to stop the execution and run the new application.

Observe the SDK Terminal window as the program executes. Play with dip switches and observe the LEDs. Notice that the system is very responsive (LEDs appear to change immediately after button is pressed).

4-4. Run the stack and Heap from BRAM

4-4-1. Go back to the linker script, and change the heap and stack to **axi_bram_ctrl_0_S_AXI_BASEADDR** memory

4-4-2. Run the program again, and observe that the program executes much slower as the stack is in BRAM. This is because access to the cached DDR in the PS is much faster than BRAM in the PL.

4-4-3. When finished, click on the **Terminate** button in the *Console* tab.

4-4-4. Close SDK and Vivado.

4-4-5. Power OFF the board.

Conclusion

Use SDK 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. SDK imports an xml file, creates a corresponding MSS file and lets you update the settings so you can develop 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 code and libraries. If needed, you can also use a linker script to target various segments in various memories. When the application is too big to fit in the internal BRAM, you can download the application in external memory and then execute the program.