

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 Zynq board. An AXI BRAM controller and associated 8KB BRAM were added in the last lab. The application will be run from the external DDR RAM but the heap and stack will be placed in BRAM by modifying the linker script for the project to place the code and data sections in DDR RAM and stack and heap section 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 Zybo board

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 Zybo board.

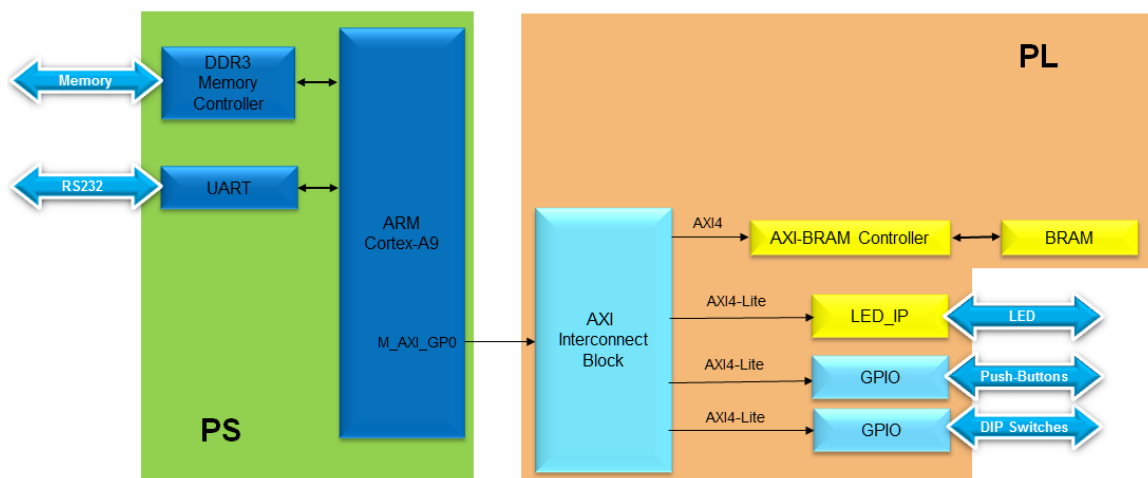
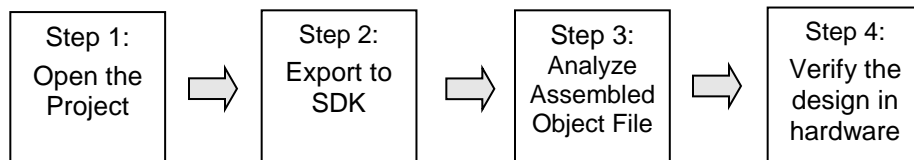


Figure 1. Design used from the Previous Lab

General Flow for this Lab



In the instructions below;

{sources} refers to: C:\xup\embedded\2015_2_zynq_sources

{labs} refers to : C:\xup\embedded\2015_2_zynq_labs

{labsolutions} for the ZedBoard refers to: C:\xup\embedded\2015_2_zedboard_labsolution
or for the Zybo refers to: C:\xup\embedded\2015_2_zybo_labsolution

Opening the Project

Step 1

1-1. Use the lab3 project from the last lab, or use the *lab3* project in the **{labsolutions}** directory, 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 {labsolutions} 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 {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.

2-1-1. Click **File > Export > Export Hardware**.

2-1-2. Click on the checkbox of *Include the bitstream* and then click **Yes** to overwrite.

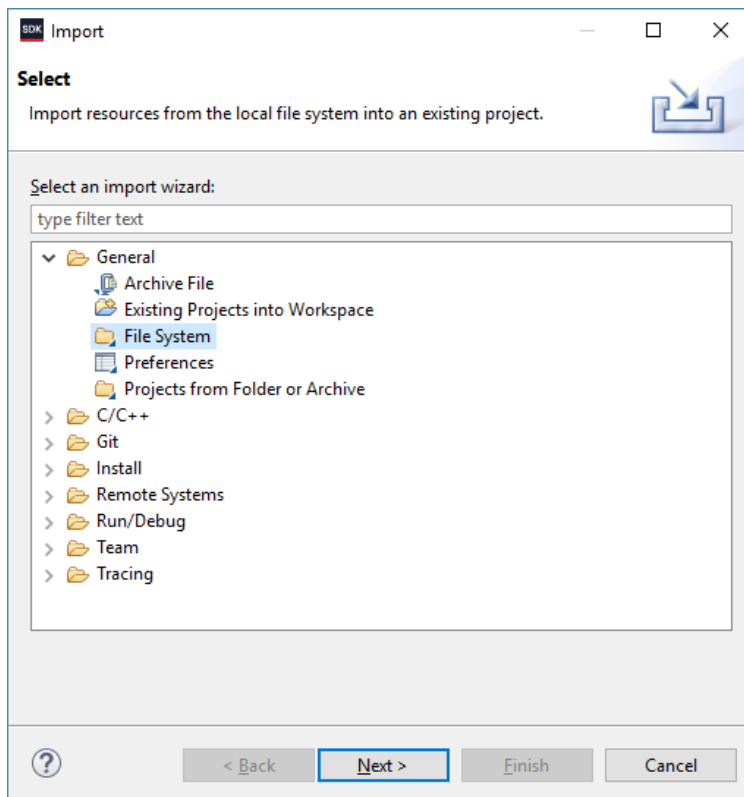
2-1-3. Select **File > Launch SDK** and click **OK**.

2-2. Close previously created projects. Create an empty project called lab4. Import lab4.c file from the {sources} directory

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**, **standalone_bsp_0**, and the **system_wrapper_hw_platform_1** projects from the previous lab, and click **Close Project**, as these projects will not be used in this lab. They can be reopened later if needed. Another option is to remove them from your work space as well as manually delete them from the .sdk folder.

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

- 2-2-3.** Enter **lab4** as the *Project Name*, and for *Board Support Package*, choose **Create New lab4_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**. Click **Next**.



- 2-2-7.** Browse to {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.** Expand **lab4_bsp** and open the **system.mss**
- 2-2-10.** Click on **Documentation** link corresponding to **buttons** 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.

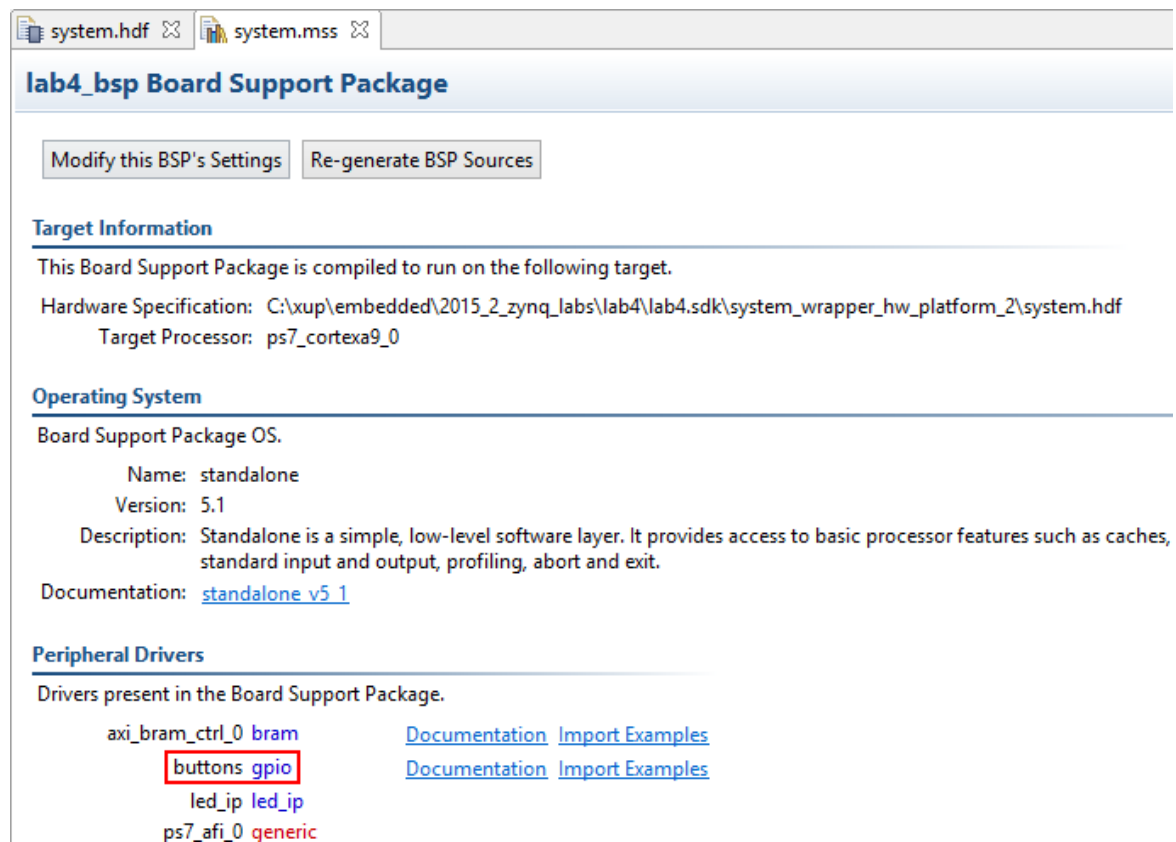


Figure 2. Accessing device driver documentation

- 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.** Double-click on **lab4.c** in the Project Explorer view to open the file. This will populate the **Outline** tab.
- 2-2-13.** Double click on **xgpio.h** in the *Outline* view and review the contents of the file to see the available function calls for the GPIO.

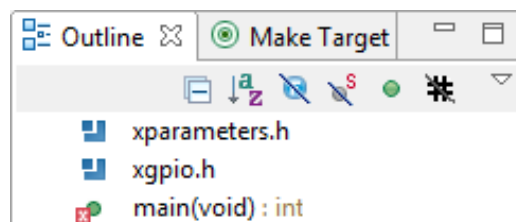


Figure 3. Outline View

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:

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.

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

2-2-14. Open the header file **xparameters.h** by double-clicking on **xparameters.h** in the **Outline** tab

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 **switches** peripheral:

#define XPAR_SWITCHES_DEVICE_ID 1

● — **Note: The number might be different**

Notice the other #define XPAR_SWITCHES* statements in this section for the switches peripheral, and in particular the address of the peripheral defined by: **XPAR_SWITCHES_BASEADDR**

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

```

1 #include "xparameters.h"
2 #include "xgpio.h"
3
4 //=====
5
6 int main (void)
7 {
8
9     XGpio dip, push;
10    int i, psb_check, dip_check;
11
12    xil_printf("-- Start of the Program --\r\n");
13
14    XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID); // Modify this
15    XGpio_SetDataDirection(&dip, 1, 0xffffffff);
16
17    XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Modify this
18    XGpio_SetDataDirection(&push, 1, 0xffffffff);
19
20
21    while (1)
22    {
23        psb_check = XGpio_DiscreteRead(&push, 1);
24        xil_printf("Push Buttons Status %x\r\n", psb_check);
25        dip_check = XGpio_DiscreteRead(&dip, 1);
26        xil_printf("DIP Switch Status %x\r\n", dip_check);
27
28        // output dip switches value on LED_ip device
29
30        for (i=0; i<9999999; i++);
31    }
32 }

```

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

2-2-16. Do the same for the *BUTTONS*; find the macro (*#define*) for the *BUTTONS* peripheral in *xparameters.h*, and modify line 17 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 *lab4_bsp* in the project view, right-click, and select **Board Support Package Settings**.

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

2-3-3. If the *led_ip* driver has not already been selected, 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	Dr...
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	2.0
axi_bram_ctrl_0	axi_bram_ctrl	bram	4.0
btms_4bit	axi_gpio	gpio	4.0
led_ip	led_ip	led_ip	1.0
ps7_afi_0	ps7_afi	generic	2.0
ps7_afi_1	ps7_afi	generic	2.0
ps7_afi_2	ps7_afi	generic	2.0
ps7_afi_3	ps7_afi	generic	2.0

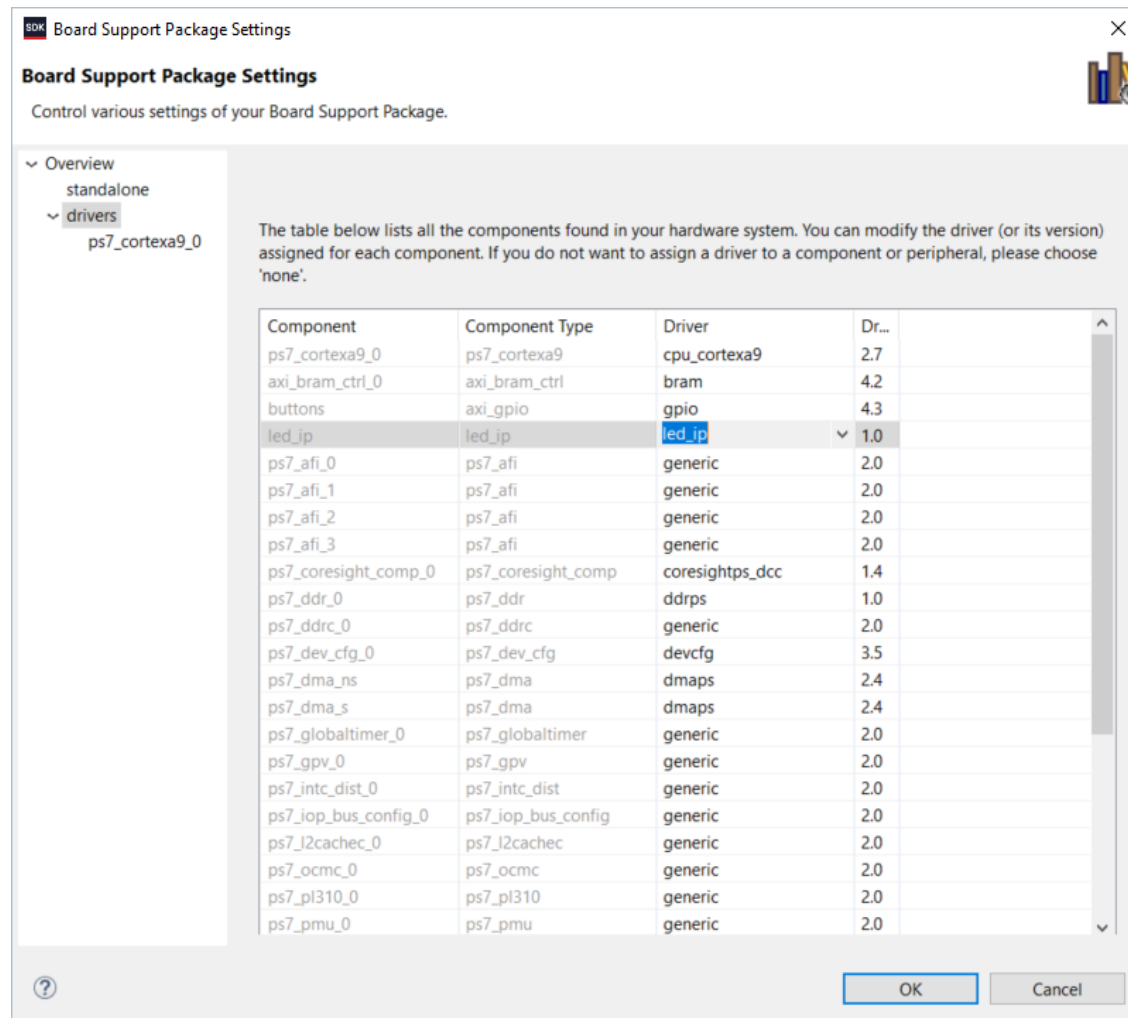


Figure 5. Assign led_ip driver

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 `led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\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 the {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_S_AXI_BASEADDR, 0, dip_check);
```

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


```

#include "xparameters.h"
#include "xgpio.h"
#include "led_ip.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_SWITCHES_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BUTTONS_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
        LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_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 > Launch Shell**.

3-1-2. Change the directory to **{Labs}\Lab4\Lab4.sdk\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 **armr5-none-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	UMA	LMA	File off	Algn
0	.text	00001b34	00100000	00100000	00008000	2××6
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.init	00000018	00101b34	00101b34	00009b34	2××2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.fini	00000018	00101b4c	00101b4c	00009b4c	2××2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
3	.rodata	0000018c	00101b64	00101b64	00009b64	2××2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.data	00000494	00101cf0	00101cf0	00009cf0	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
	CONTENTS, ALLOC, LOAD, DATA					
9	.ARM.attributes	00000033	0010800c	0010800c	0001000c	2××0
	CONTENTS, READONLY					
10	.bss	0000002c	0010800c	0010800c	0001000c	2××2
	ALLOC					
11	.heap	00002008	00108038	00108038	0001000c	2××0
	ALLOC					
12	.stack	00003800	0010a040	0010a040	0001000c	2××0
	ALLOC					

Figure 7. Object dump results - .text, .stack, and .heap in the DDR3 space


Verify in Hardware

Step 4

4-1. Connect the board with micro-usb cable(s) and power it ON. Establish the serial communication using SDK's Terminal tab.

4-1-1. Make sure that micro-USB cable(s) is(are) connected between the board and the PC. Turn ON the power.

4-1-2. Select the  **Terminal** tab. If it is not visible then select **Window > Show view > Terminal**.

4-1-3. Click on  and if required, select appropriate COM port (depends on your computer), and configure it with the parameters as shown. (These settings may have been saved from previous lab).

4-2. Program the FPGA by selecting Xilinx Tools > Program FPGA and assigning system_wrapper.bit file. Run the TestApp application and verify the functionality.

4-2-1. Select **Xilinx Tools > Program FPGA**.

4-2-2. Click the **Program** button to program the FPGA.

- 4-2-3.** Select **lab4** in *Project Explorer*, right-click and select **Run As > Launch on Hardware (GDB)** 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.

```
DIP Switch Status C
Push Buttons Status 0
DIP Switch Status C
Push Buttons Status 0
DIP Switch Status C
Push Buttons Status 0
DIP Switch Status C
Push Buttons Status 0
DIP Switch Status C
```

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.

- 4-3. Change the linker script to target Code sections to the BRAM controller and objdump lab4.elf and look at the sections it has created.**

- 4-3-1.** Right click on `lab4` and click **Generate Linker Script...**

Note that all four major sections, code, data, stack and heap are by default assigned to DDR controller. We need to change Heap and Stack to use BRAM.

- 4-3-2.** In the *Basic Tab* leave the *Code* and *Data* sections to `ps7_ddr_0_S_AXI_BASEADDR`, and change the *Heap and Stack* in section to `axi_bram_ctrl_0_S_AXI_BASEADDR` memory and click **Generate**, and click **Yes** to overwrite.

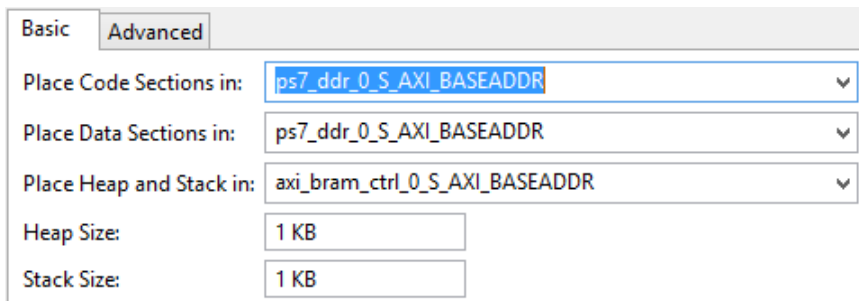


Figure 9. Targeting Stack/Heap sections to BRAM

The program will compile again.

- 4-3-3.** Type `armr5-none-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	UMA	LMA	File off	Algn
0	.text	00001b34	00100000	00100000	00008000	2xx6
1	.init	00000018	00101b34	00101b34	00009b34	2xx2
2	.fini	00000018	00101b4c	00101b4c	00009b4c	2xx2
3	.rodata	0000018c	00101b64	00101b64	00009b64	2xx2
4	.data	00000494	00101cf0	00101cf0	00009cf0	2xx3
5	.eh_frame	00000004	00102184	00102184	0000a184	2xx2
6	.mmu_tbl	00004000	00104000	00104000	0000c000	2xx0
7	.init_array	00000008	00108000	00108000	00010000	2xx2
8	.fini_array	00000004	00108008	00108008	00010008	2xx2
9	.ARM.attributes	00000033	0010800c	0010800c	0001000c	2xx0
10	.bss	0000002c	0010800c	0010800c	0001000c	2xx2
11	.heap	00000400	40000000	40000000	00018000	2xx0
12	.stack	00001c00	40000400	40000400	00018000	2xx0

Figure 10. The .heap and .stack sections targeted to BRAM whereas the rest of the application is in DDR

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

- 4-4-1. Select **lab4** in *Project Explorer*, right-click and select **Run As > Launch on Hardware (GDB)** 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 relatively slow in displaying the message in the Terminal tab and to change in the switches as the stack and heap are from a non-cached BRAM memory.

- 4-4-2. When finished, click on the **Terminate** button in the *Console* tab.

- 4-4-3. Exit SDK and Vivado.

- 4-4-4. 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 hdf 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.