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 Zed board. An AXI BRAM controller and 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 Zed 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:

1. You will open the Vivado project, export to and invoke SDK,
2. create a software project,
3. analyze assembled object files
4. 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. ***<u>A basic software application will be developed</u>*** to access the LEDs on the Zed board.

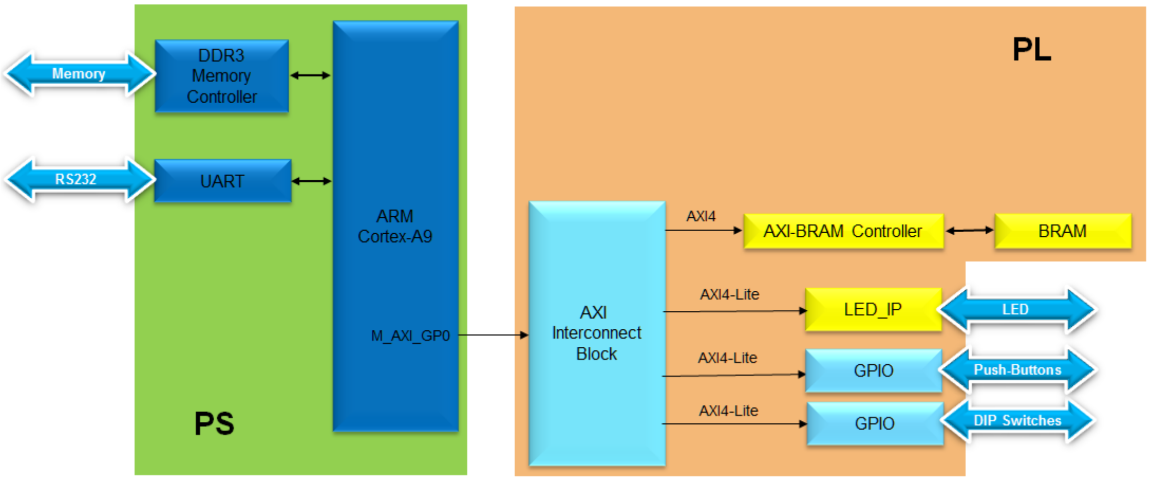
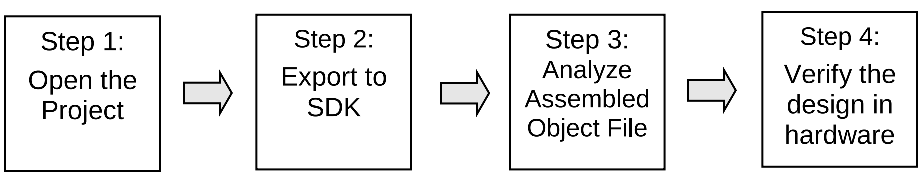


Figure 1. Design used from the Previous Lab

General Flow for this Lab



In the instructions below;
{**sources**} refers to: C:\Xilinx_trn\Zynq_base\lab_sources
{**labs**} refers to : C:\Xilinx_trn\Zynq_base

Opening the Project Step 1

1-1. Use the lab3 project from the last lab and save it as *lab4*

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

1-1-2. Select **File > Project > Save 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:/Xilinx_trn/Zynq_base* 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.**

2-1-1.1. Click on the checkbox of *Include the bitstream*

2-1-1.2. Click **OK** and then click **Yes** to overwrite.

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

2-2. In SDK: 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,

2-2-1.1. Select (by Click+Cntr): **standalone_bsp_0, system_wrapper_hw_platform_1, TestApp** projects from the previous lab

2-2-1.2. Right click and select **Close Project**.

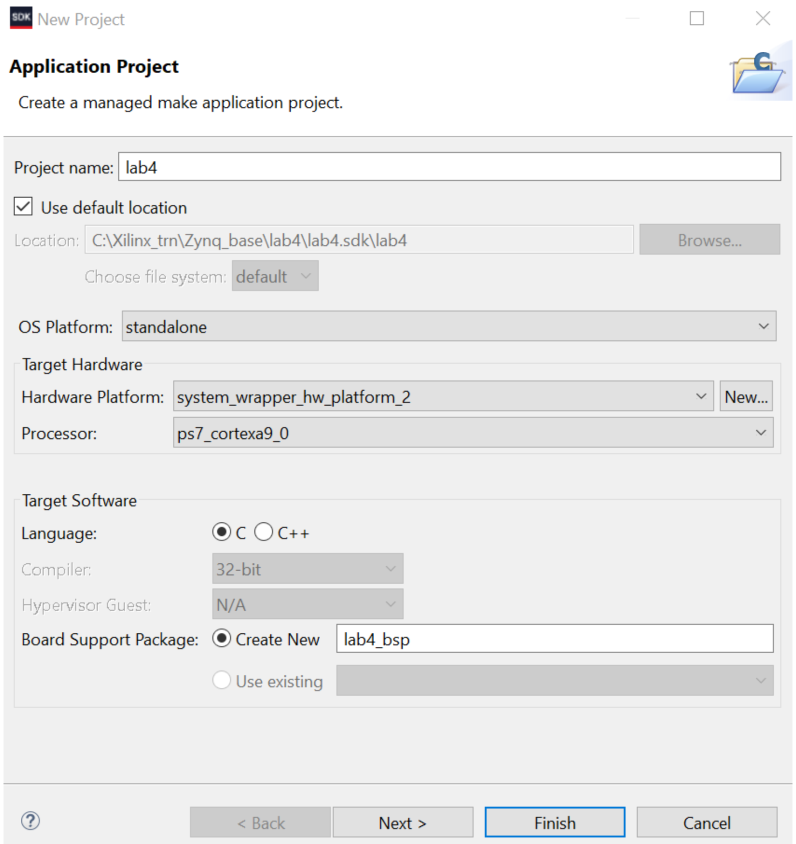
These projects will not be used in this lab. They can be reopened later if needed.

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

2-2-3. In the window appeared:

2-2-3.1. *Project Name:* set as **lab4**.

2-2-3.2. *Board Support Package:* set **Create New** and be sure that a name is **lab4_bsp**.



2-2-3.3. Click **Next**.

2-2-3.4. In the window appeared select *Empty Application* and click **Finish.**

2-2-4. Expand **lab4** in the project view and right-click in the *src folder* and select **Import.**

2-2-5. Expand **General** category and double-click on **File System.**

2-2-6. Browse to **C:\Xilinx_trn\Zynq_base\lab_sources\lab4** folder and click OK.

2-2-7. Select **lab4.c** and click **Finish** to add the file to the project. (Ignore any errors for now).

2-2-8. Expand **lab4_bsp** and open the **system.mss**

2-2-9. 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.

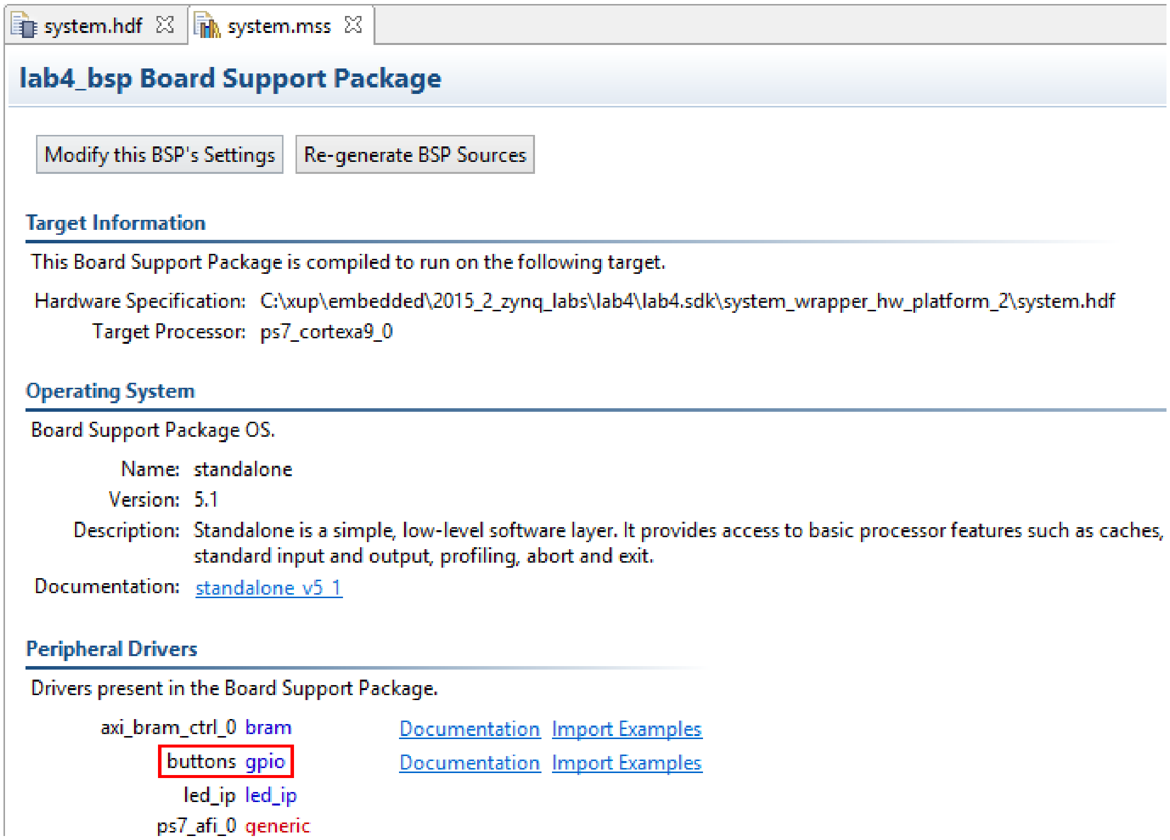


Figure 2. Accessing device driver documentation

2-2-10. View the various C and Header files associated with the GPIO by clicking **File List** at the top of the page.

2-2-11. Double-click on **lab4.c** in the Project Explorer view to open the file.

*This will populate the **Outline** tab.*

2-2-12. 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.

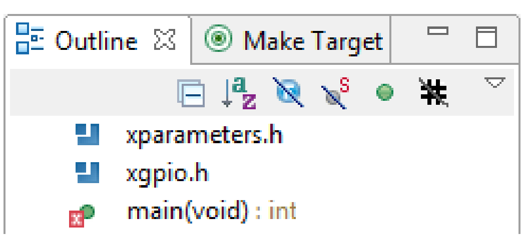


Figure 3. Outline View

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)

gpio_v4_3

Xilinx SDK Drivers API Documentation

Overview | Data Structures | APIs | File List | Examples

- gpio_v4_3
  - Data Structures
  - APIs
  - File List
    - xgpio.c
    - xgpio.h
      - XGpio_CfgInitialize
      - XGpio_DiscreteClear
      - XGpio_DiscreteRead
      - XGpio_DiscreteSet
      - XGpio_DiscreteWrite
      - XGpio_GetDataDirection
      - XGpio_Initialize
      - XGpio_InterruptClear
      - XGpio_InterruptDisable
      - XGpio_InterruptEnable
      - XGpio_InterruptGetEnabled
      - XGpio_InterruptGetStatus
      - XGpio_InterruptGlobalDisable
      - XGpio_InterruptGlobalEnable
      - XGpio_LookupConfig
      - XGpio_SelfTest
      - XGpio_SetDataDirection
    - xgpio_example.c
    - xgpio_extra.c
    - xgpio_g.c
    - xgpio_i.h
    - xgpio_intr.c
    - xgpio_intr_tapp_example.c
    - xgpio_l.h
    - xgpio_low_level_example.c
    - xgpio_selftest.c
    - xgpio_sinit.c

◆ XGpio_Initialize()

```
int XGpio_Initialize ( XGpio * InstancePtr,
                       u16     DeviceId
                     )
```

```
#include <xgpio.h>
```

Initialize the **XGpio** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters

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 instance/driver through the **XGpio** API must be made with this pointer.
DeviceId	is the unique id of the device controlled by this **XGpio** instance. Passing in a device id associates the generic **XGpio** instance to a specific device, as chosen by the caller or application developer.

Returns

- XST_SUCCESS if the initialization was successfull.
  - XST_DEVICE_NOT_FOUND if the device configuration data was not found for a device with the supplied device ID.

Note

None.

References **XGpio_CfgInitialize()**, and **XGpio_LookupConfig()**.

Referenced by **GpioInputExample()**, **GpioOutputExample()**, and **main()**.

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. Open lab4.c file

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.

2-2-15. Find #define used to identify the **switches** peripheral:

#define XPAR_SWITCHES_DEVICE_ID 1

Note: The ID number might be different

2-2-16. 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`**

2-2-17. Modify line 15 of lab4.c to use XPAR_SWITCHES_DEVICE_ID in XGpio_Initialize function

```
15    XGpio_Initialize(&switches, XPAR_SWITCHES_DEVICE_ID); // Modify this
```

2-2-18. Find the macro (#define) for the *BUTTONS* peripheral in xparameters.h,

2-2-19. Modify line 18 in lab4.c to use XPAR_BUTTONS_DEVICE_ID in XGpio_Initialize function

```
18    XGpio_Initialize(&buttons, XPAR_BUTTONS_DEVICE_ID); // Modify this
```

2-2-20. Save the file lab4.c. The project will be rebuilt.

If there are any errors, check and fix your code.

2-2-21. Your C code will eventually read the value of the switches and buttons and output it to the console (`xil_printf` function).

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


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

int main (void)
{

    XGpio switches, buttons;
    int buttons_check=0, buttons_check_old=0, switches_check=0, switches_check_old=0;

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

    XGpio_Initialize(&switches, XPAR_SWITCHES_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&switches, 1, 0xffffffff);

    XGpio_Initialize(&buttons, XPAR_BUTTONS_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&buttons, 1, 0xffffffff);


    while (1)
    {
        buttons_check = XGpio_DiscreteRead(&buttons, 1);
        if (buttons_check_old != buttons_check){
            xil_printf("Buttons Status %x\r\n", buttons_check);
            buttons_check_old = buttons_check;
        }
        switches_check = XGpio_DiscreteRead(&switches, 1);

        if (switches_check_old != switches_check){
            xil_printf("Switches Status %x\r\n", switches_check);
            switches_check_old = switches_check;
        }

        // output dip switches value on LED_ip device


    }
}
```

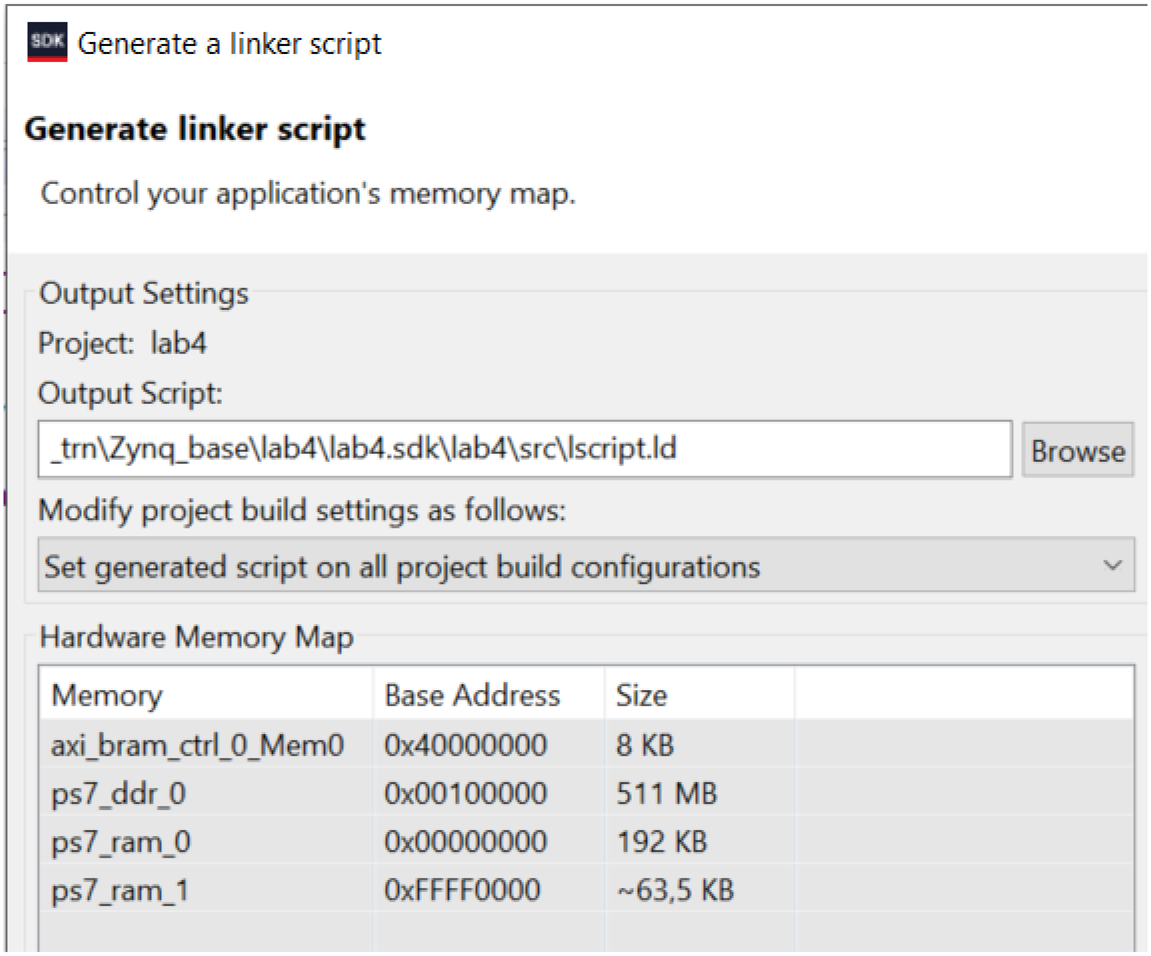
Figure 4. Completed source file with output to console only

2-3. Change the linker script to target ***code***, ***data***, ***stack*** and ***heap*** sections to the DDR controller and look at objdump lab4.elf to check the sections it has created.

2-3-1. Right click on lab4 in *Project Explorer* and click **Generate Linker Script…**

Note that **Hardware Memory Map** pane highlights all available memory resources:

- **axi_bram_cntr_0_Mem()** – BRAM memory
- **ps7_ddr_0** – DDR memory
- **ps7_ram_0 and ps7_ram_1** – embedded RAM memory



2-3-2. Assign all four major sections: *code, data, stack and heap* to DDR controller.

2-3-2.1. In the *Basic Tab* change the *Code*, *Data, Heap and Stack* sections to **ps7_ddr_0**

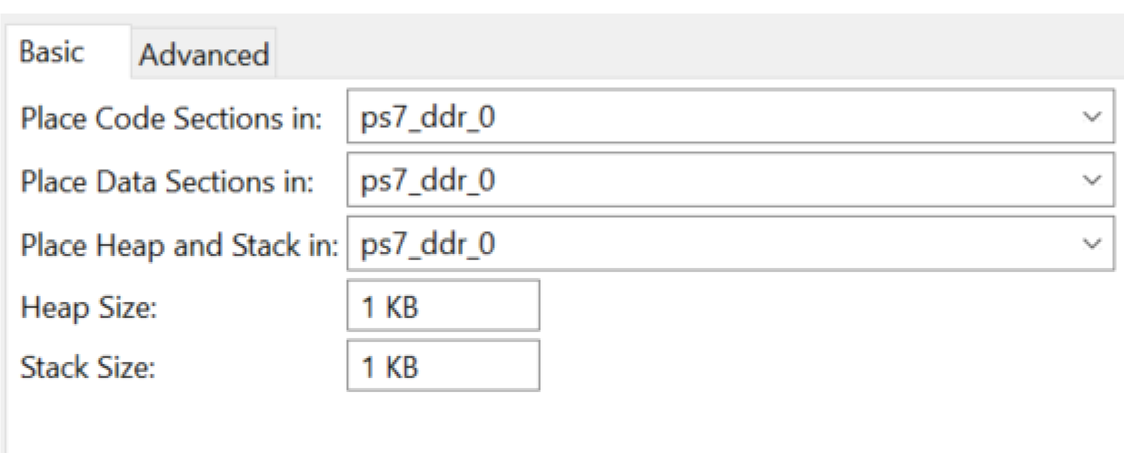


Figure 9. Targeting Stack/Heap sections to DDR

2-3-2.2. Click **Generate**, and click **Yes** to overwrite. The program will compile again.

2-3-3. Launch Shell and look at the lab4.elf sections.

2-3-3.1. Launch the shell from SDK by selecting **Xilinx > Launch Shell.**

2-3-3.2. Change the current directory *C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk* to *C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk**\lab4\Debug*** using the **cd** command in the shell.

```
C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk>cd lab4\Debug

C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk\lab4\Debug>
```

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

2-3-3.3. In the shell window type ***arm-none-eabi-objdump –h lab4.elf*** to list various sections of the lab4.elf, along with the starting address and size of each section

You should see results similar to that below:

```
C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk>cd lab4\Debug

C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk\lab4\Debug>arm-none-eabi-objdump -h lab4.elf

lab4.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019c4  00100000  00100000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         00000018  001019c4  001019c4  000119c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         00000018  001019dc  001019dc  000119dc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000184  001019f4  001019f4  000119f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000498  00101b78  00101b78  00011b78  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  00102010  00102010  00012010  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .mmu_tbl      00004000  00104000  00104000  00014000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .init_array   00000004  00108000  00108000  00018000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .fini_array   00000004  00108004  00108004  00018004  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .ARM.attributes 00000033  00108008  00108008  00018008  2**0
                  CONTENTS, READONLY
 10 .bss          00000030  00108008  00108008  00018008  2**2
                  ALLOC
 11 .heap         00000408  00108038  00108038  00018008  2**0
                  ALLOC
 12 .stack        00001c00  00108440  00108440  00018008  2**0
                  ALLOC
 13 .comment      00000031  00000000  00000000  0001803b  2**0
                  CONTENTS, READONLY
```

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

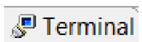
Verify in Hardware Step 3

3-1. Connect the Zed board with two micro-usb cables and power it ON. Establish the serial communication using SDK's Terminal tab.

3-1-1. Make sure that micro-USB cables are connected between the board and the PC.

3-1-2. Set all switches on the board to state Zero (switch those down).

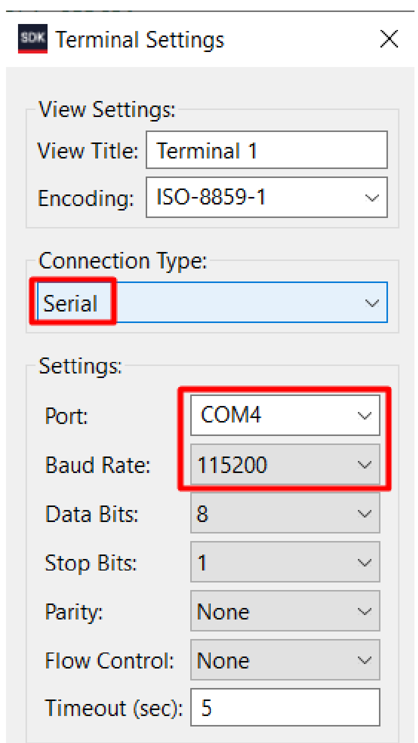
3-1-3. Turn ON the power.

3-1-4. Select the 

tab.

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

3-1-5. 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).



3-2. Program the FPGA in SDK, run the TestApp application and verify the functionality.

3-2-1. In SDK select **Xilinx** > **Program FPGA.**

3-2-2. Click the **Program** button to program the FPGA.

3-2-3. 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

3-2-4. Flip the switches/push the buttons and verify that you see the results in SDK Terminal.

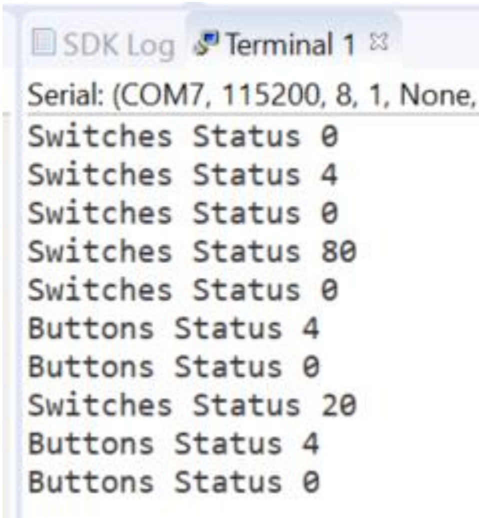


Figure 8. Switches and buttons settings displayed in SDK terminal

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

3-3. Change the linker script to target all sections to the RAM controller and look at the objdump lab4.elf sections.

3-3-1. Right click on lab4 in *Project Explorer* and click **Generate Linker Script…**

3-3-2. Assign all four major sections: *Code*, *Data, Heap, Stack* to **ps7_ram_0** memory.

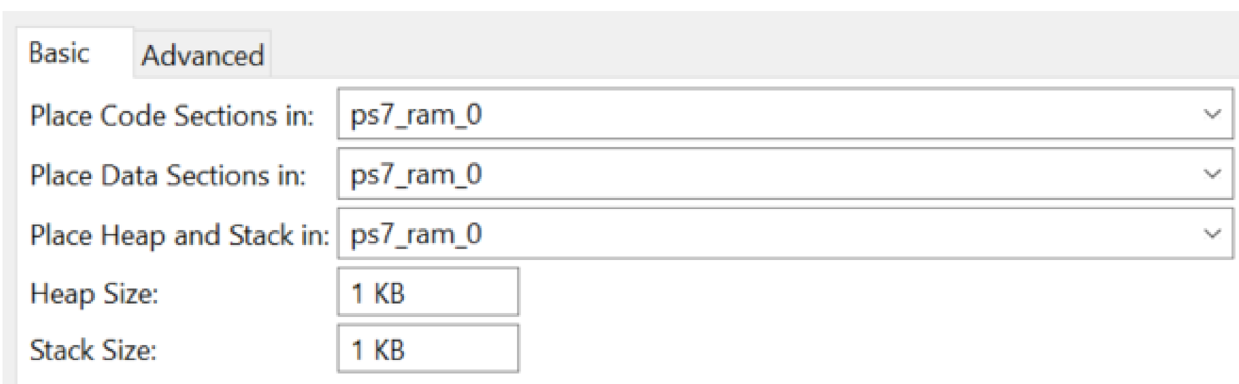


Figure 9. Targeting Stack/Heap sections to RAM

3-3-3. Click **Generate**, and click **Yes** to overwrite. The program will compile again.

3-3-4. In the shell window type ***arm-none-eabi-objdump –h lab4.elf*** to list various sections of the lab4.elf, along with the starting address and size of each section

You should see results similar to that below:

```
C:\Xilinx_trn\Zynq_base\lab4\lab4.sdk\lab4\Debug>arm-none-eabi-objdump -h lab4.elf

lab4.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019c4  00000000  00000000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         00000018  000019c4  000019c4  000119c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         00000018  000019dc  000019dc  000119dc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000184  000019f4  000019f4  000119f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000498  00001b78  00001b78  00011b78  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  00002010  00002010  00012010  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .mmu_tbl      00004000  00004000  00004000  00014000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .init_array   00000004  00008000  00008000  00018000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  8 .fini_array   00000004  00008004  00008004  00018004  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .ARM.attributes 00000033  00008008  00008008  00018008  2**0
                  CONTENTS, READONLY
 10 .bss          00000030  00008008  00008008  00018008  2**2
                  ALLOC
 11 .heap         00000408  00008038  00008038  00018008  2**0
                  ALLOC
 12 .stack        00001c00  00008440  00008440  00018008  2**0
                  ALLOC
 13 .comment      00000031  00000000  00000000  0001803b  2**0
```

Figure 10. The ,heap and .stack sections targeted to RAM

3-4. Execute the lab4.elf application and observe the application working.

3-4-1. 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

3-4-2. Flip the switches/push the buttons and verify that you see the results in SDK Terminal.

3-5. Assign the led_ip driver from the *driver* directory to the led_ip instance.

3-5-1. Select **lab4_bsp** in the *Project Explorer*, right-click, and select **Board Support Package Settings.**

3-5-2. Select *drivers* on the left (under *Overview*)

3-5-3. 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.**

Component	Component Type	Driver	Dri...
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9	2.0
axi_bram_ctrl_0	axi_bram_ctrl	bram	4.0
btns_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

3-5-4. Click **OK.**

3-6. 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.

3-6-1. In SDK select **File > Open File** and, in windows explorer, browse to C:\Xilinx_trn\Zynq_base*led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src*

3-6-2. Notice the files in this directory and open led_ip.c.

This file only includes the header file for the IP.

3-6-3. 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_In32( ). 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.

3-6-4. Modify your C code to echo the dip switch settings on the LEDs by using the **led_ip** driver API macros, and save the application.

3-6-4.1. Open lab4.c file.

3-6-4.2. Include the header file in line 3 of the source code in lab4.c file:

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

3-6-4.3. Include the function to write to the IP in line 37 of the source code in lab4.c file:

```
LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, switches_check);
```

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

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

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

int main (void)
{

    XGpio switches, buttons;
    int buttons_check, buttons_check_old, switches_check, switches_check_old;

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

    XGpio_Initialize(&switches, XPAR_SWITCHES_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&switches, 1, 0xffffffff);

    XGpio_Initialize(&buttons, XPAR_BUTTONS_DEVICE_ID); // Modify this
    XGpio_SetDataDirection(&buttons, 1, 0xffffffff);


    while (1)
    {
        buttons_check = XGpio_DiscreteRead(&buttons, 1);
        if (buttons_check_old != buttons_check){
            xil_printf("Buttons Status %x\r\n", buttons_check);
            buttons_check_old = buttons_check;
        }
        switches_check = XGpio_DiscreteRead(&switches, 1);

        if (switches_check_old != switches_check){
            xil_printf("Switches Status %x\r\n", switches_check);
            switches_check_old = switches_check;
        }

        // output dip switches value on LED_ip device
        LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, switches_check);

    }
}
```

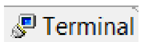
Figure 6. The completed C file

3-6-5. Save the file and the program will be compiled again.

Verify in Hardware Step 4

4-1. Connect the Zed board with micro-usb cables 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 

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 in SDK, run the TestApp application and verify the functionality.

4-2-1. In SDK select **Xilinx** > **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 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 switches and push buttons will change the results displayed.

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

4-3-1. Right click on lab4 in *Project Explorer* and click **Generate Linker Script…**

Note that all four major sections, code, data, stack and heap are to be assigned to RAM controller (ps7_ram_0).

4-3-2. In the *Basic Tab* change the *Heap and Stack in* section to **axi_bram_ctrl_0_Mem** memory and click **Generate**, and click **Yes** to overwrite.

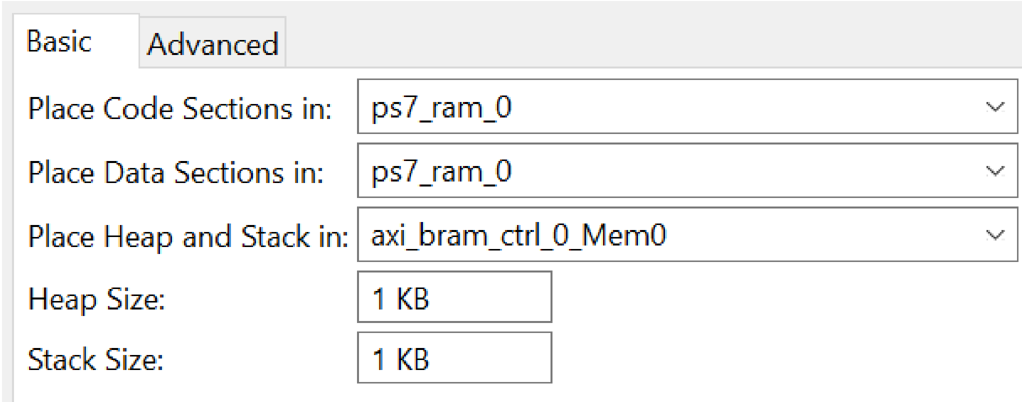


Figure 9. Targeting Stack/Heap sections to BRAM

The program will compile again.

4-3-3. Type ***arm-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:

```
:\Xilinx_trn\Zynq_base\lab4\lab4.sdk\lab4\Debug>arm-none-eabi-objdump -h lab4.elf

ab4.elf:     file format elf32-littlearm

ections:
dx Name          Size      VMA       LMA       File off  Algn
 0 .text         00001a04  00000000  00000000  00010000  2**6
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .init         00000018  00001a04  00001a04  00011a04  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .fini         00000018  00001a1c  00001a1c  00011a1c  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .rodata       00000184  00001a34  00001a34  00011a34  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data         00000498  00001bb8  00001bb8  00011bb8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 5 .eh_frame     00000004  00002050  00002050  00012050  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .mmu_tbl      00004000  00004000  00004000  00014000  2**0
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .init_array   00000004  00008000  00008000  00018000  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 8 .fini_array   00000004  00008004  00008004  00018004  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 9 .ARM.attributes 00000033  00008008  00008008  00018008  2**0
                 CONTENTS, READONLY
10 .bss          00000030  00008008  00008008  00018008  2**2
                 ALLOC
11 .heap         00000400  40000000  40000000  00020000  2**0
                 ALLOC
12 .stack        00001c00  40000400  40000400  00020000  2**0
                 ALLOC
```

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

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 switches and observe the LEDs. Notice that the system can be 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. Exit SDK and Vivado.

4-4-3. 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.