LAB 4.

Writing Basic Software Applications

Introduction

This lab guides you through the process of writing a basic software application. The software will write to the LEDs on the Spartan-3E starter kit. You will add an XPS BRAM controller and modify the linker script to place the text section in the BRAM. Finally, you will verify that the design operates as expected in hardware.

Objectives

After completing this lab, you will be able to:

- Add an internal Block RAM memory controller
- Write a basic application to access an IP peripheral in SDK
- Develop a linker script
- Partition the executable sections onto both the LMB and PLB memory spaces
- Generate a bit file
- Download the bit file and verify on the Spartan-3E starter kit

Design Description

The purpose of this lab exercise is to Extend the processor system created in lab3 by adding a memory controller (see Figure 1) and the writing a basic software application to access the LEDs on the Spartan-3E starter kit.

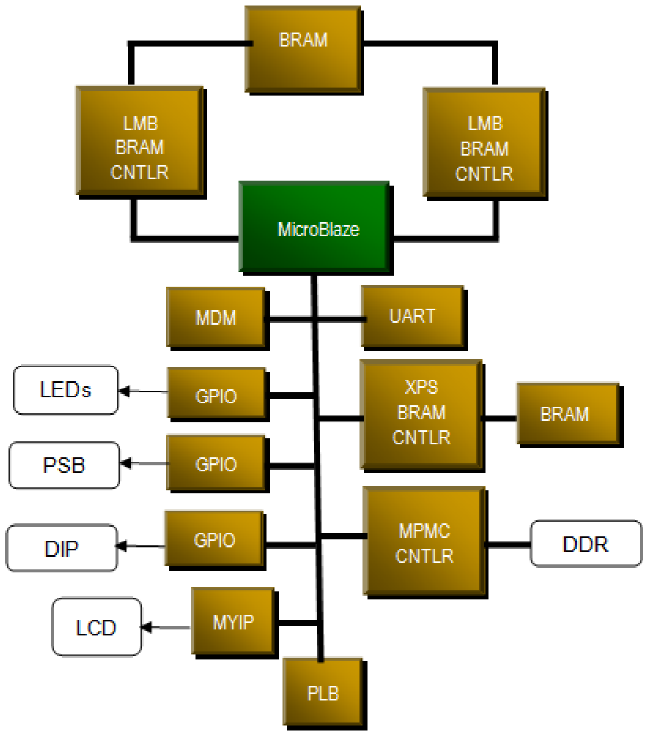


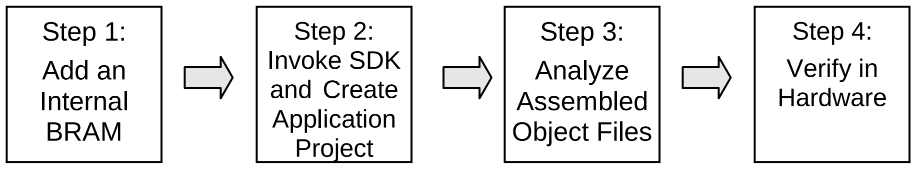
Figure 1. Design Updated from Previous Lab

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 add an internal BRAM, invoke SDK and create application project, analyze assembled object files and, finally, verify in hardware.
Note: If you are unable to complete the lab at this time, you can download the original lab files for this module from the Xilinx University Program site at http://www.xilinx.com/university

General Flow for this Lab


Step 1: Add an Internal BRAM
Step 2: Invoke SDK and Create Application Project
Step 3: Analyze Assembled Object Files
Step 4: Verify in Hardware

Add an Internal BRAM Step 1

1-1. Create a *lab4* folder and copy the contents of the *lab3* folder into the *lab4* folder. Launch Xilinx Platform Studio (XPS) and open the project file.

1-1-1. Create a *lab4* folder in the **D:\** directory and copy the contents from *lab3* to *lab4*.

1-1-2. Open XPS by selecting **Start → All Programs → Xilinx → Xilinx ISE Design Suite 13.2 → EDK → Xilinx Platform Studio.**

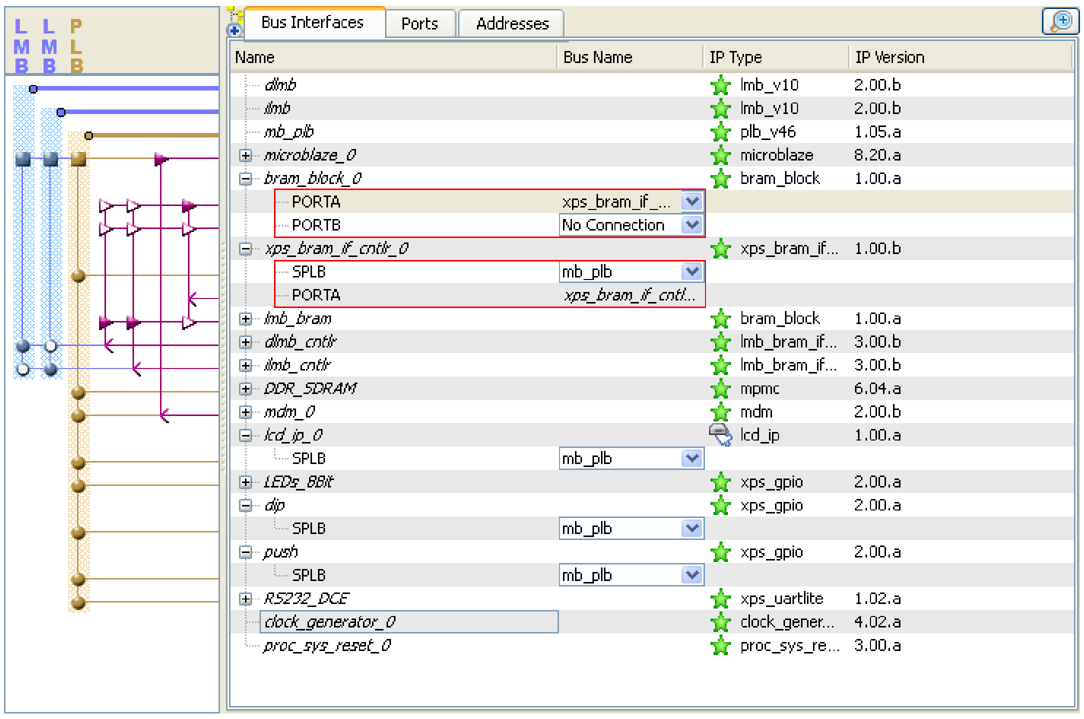
1-1-3. Browse to the *lab4* directory and open the project **system.xmp.**

1-2. Add a BRAM controller and BRAM to the design.

1-2-1. From the IP catalog, add the following IP to the embedded hardware design, accepting the default settings:

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

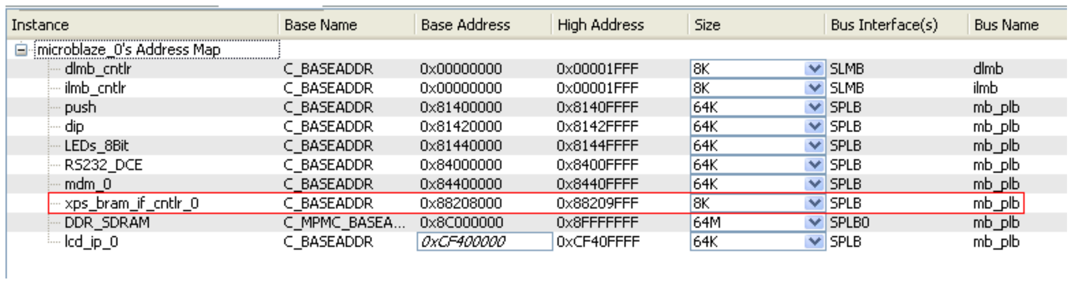
1-2-2. Connect the BRAM controller to the PLB and connect the BRAM to the BRAM controller (see **Figure 2**).



Name	Bus Name	IP Type	IP Version
dlmb		lmb_v10	2.00.b
ilmb		lmb_v10	2.00.b
mb_plb		plb_v46	1.05.a
microblaze_0		microblaze	8.20.a
bram_block_0		bram_block	1.00.a
PORTA	xps_bram_if_...		
PORTB	No Connection		
xps_bram_if_cntlr_0		xps_bram_if...	1.00.b
SPLB	mb_plb		
PORTA	xps_bram_if_cntl...		
lmb_bram		bram_block	1.00.a
dlmb_cntlr		lmb_bram_if...	3.00.b
ilmb_cntlr		lmb_bram_if...	3.00.b
DDR_SDRAM		mpmc	6.04.a
mdm_0		mdm	2.00.b
lcd_ip_0		lcd_ip	1.00.a
SPLB	mb_plb		
LEDs_8Bit		xps_gpio	2.00.a
dip		xps_gpio	2.00.a
SPLB	mb_plb		
push		xps_gpio	2.00.a
SPLB	mb_plb		
RS232_DCE		xps_uartlite	1.02.a
clock_generator_0		clock_gener...	4.02.a
proc_sys_reset_0		proc_sys_re...	3.00.a

Figure 2. Add internal XPS memory controller

1-2-3. Select a size of **8K** for the XPS BRAM controller under Unmapped Addresses in Addresses tab, and click **Generate Addresses.**



Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Bus Name
microblaze_0's Address Map						
dlmb_cntlr	C_BASEADDR	0x00000000	0x00001FFF	8K	SLMB	dlmb
ilmb_cntlr	C_BASEADDR	0x00000000	0x00001FFF	8K	SLMB	ilmb
push	C_BASEADDR	0x81400000	0x8140FFFF	64K	SPLB	mb_plb
dip	C_BASEADDR	0x81420000	0x8142FFFF	64K	SPLB	mb_plb
LEDs_8Bit	C_BASEADDR	0x81440000	0x8144FFFF	64K	SPLB	mb_plb
RS232_DCE	C_BASEADDR	0x84000000	0x8400FFFF	64K	SPLB	mb_plb
mdm_0	C_BASEADDR	0x84400000	0x8440FFFF	64K	SPLB	mb_plb
xps_bram_if_cntlr_0	C_BASEADDR	0x88208000	0x88209FFF	8K	SPLB	mb_plb
DDR_SDRAM	C_MPMC_BASEA...	0x8C000000	0x8FFFFFFF	64M	SPLB0	mb_plb
lcd_ip_0	C_BASEADDR	0xCF400000	0xCF40FFFF	64K	SPLB	mb_plb

Figure 3. Specify Size and Address of Internal BRAM Memory Controller

1-2-4. Generate hardware bitstream by clicking **Hardware → Generate Bitstream** (this is to complete the hardware flow in XPS before we start the software development flow in SDK.

Invoke SDK and Create Application Project Step 2

2-1. Start SDK from XPS, generate software platform project with default settings and default software project name.

Skip steps starting from 2-1-3 to 2-2 and remove lab3.c, if you were continuing with lab3.

2-1-1. Start SDK by clicking **Project → Export Hardware Design to SDK** ...

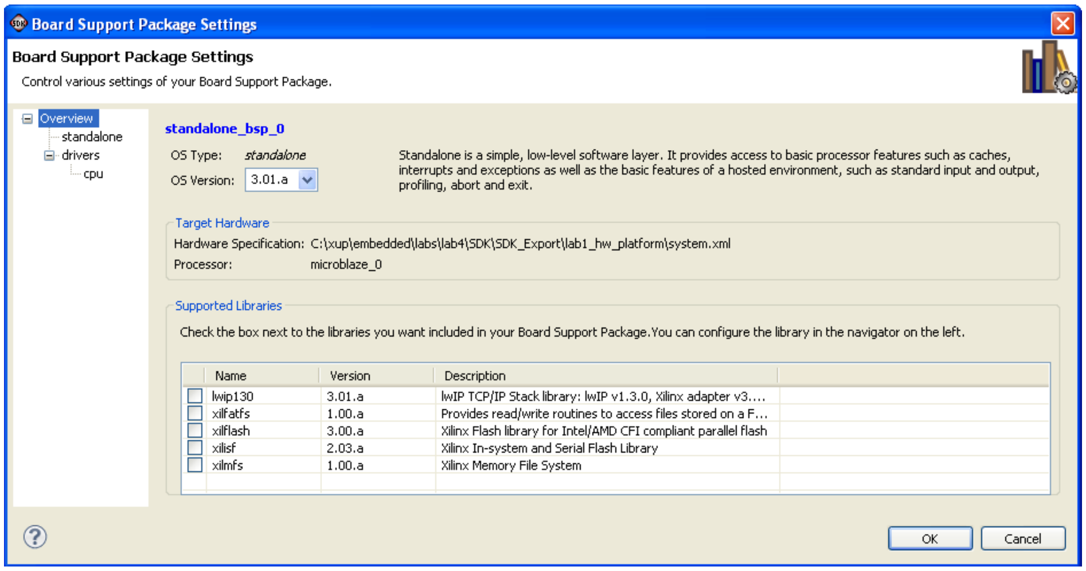
2-1-2. Click on **Export & Launch SDK** button with default settings and browse to **D:\lab3\SDK\SDK_Export**.

Skip steps starting from 2-1-3 to 2-2 and remove lab3.c, if you were continuing with lab3.

2-1-3. In SDK, select **File → New → Xilinx Board Support Package.**

2-1-4. Click **Finish** with default settings (with standalone operating system).

This will open the Software Platform Settings form showing the OS and libraries selections



Board Support Package Settings

Control various settings of your Board Support Package.

standalone_bsp_0

OS Type: standalone

OS Version: 3.01.a

Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.

Target Hardware

Hardware Specification: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system.xml

Processor: microblaze_0

Supported Libraries

Check the box next to the libraries you want included in your Board Support Package.You can configure the library in the navigator on the left.

	Name	Version	Description
☐	lwip130	3.01.a	lwIP TCP/IP Stack library: lwIP v1.3.0, Xilinx adapter v3....
☐	xilfatfs	1.00.a	Provides read/write routines to access files stored on a F...
☐	xilflash	3.00.a	Xilinx Flash library for Intel/AMD CFI compliant parallel flash
☐	xilisf	2.03.a	Xilinx In-system and Serial Flash Library
☐	xilmfs	1.00.a	Xilinx Memory File System

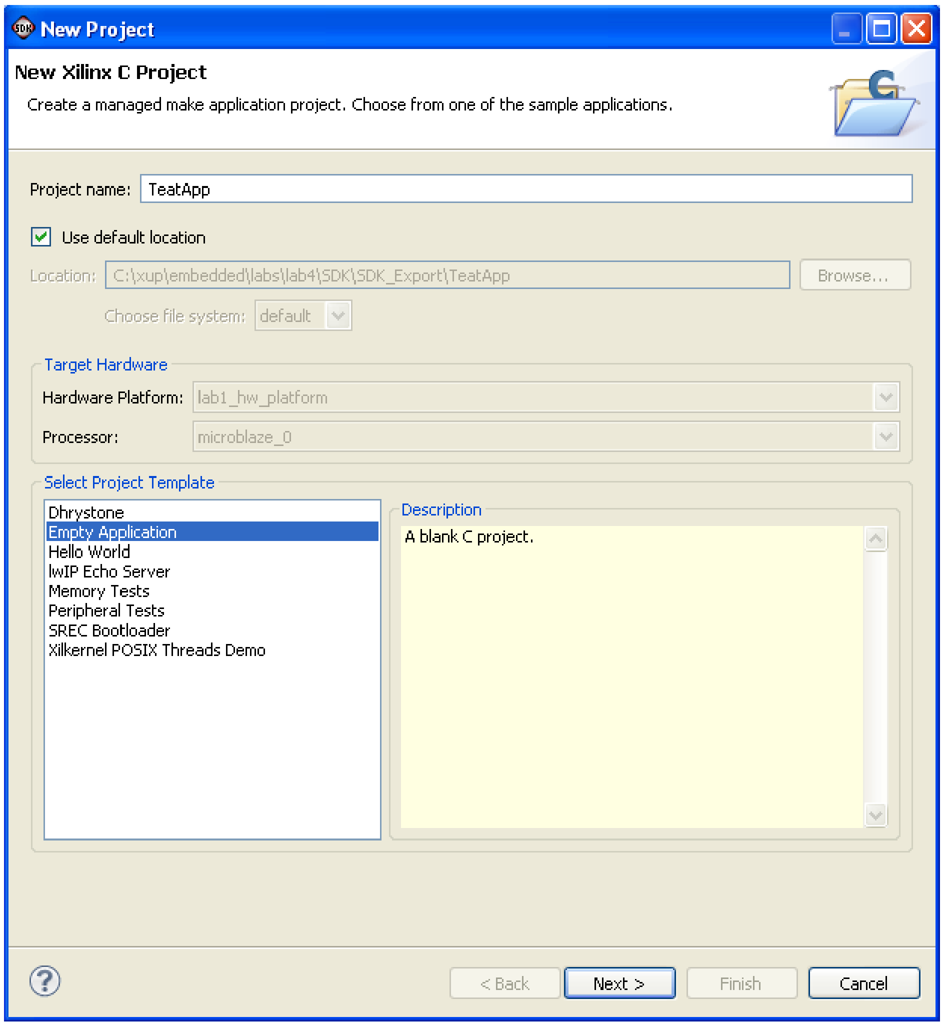
Figure 4. Board Support Package Settings

2-1-5. Click **OK** to accept the default settings, as we want to create a **standalone_bsp_0** software platform project without requiring any additional libraries support.

The library generator will run in the background and will create **xparameters.h** file in the **D:\lab4\standalone_bsp_0\microblaze_0\include\** directory.

2-1-6. Click on **File** in the Xilinx SDK window and select **New →Xilinx C Project.**

2-1-7. Select **Empty Application** in the *Select Project Template* window, and enter **TestApp** as the Project Name and click **Next.**


New Project

New Xilinx C Project

Create a managed make application project. Choose from one of the sample applications.

Project name: TeatApp

☑ Use default location

Location: C:\xup\embedded\labs\lab4\SDK\SDK_Export\TeatApp

Choose file system: default

Target Hardware

Hardware Platform: lab1_hw_platform

Processor: microblaze_0

Select Project Template

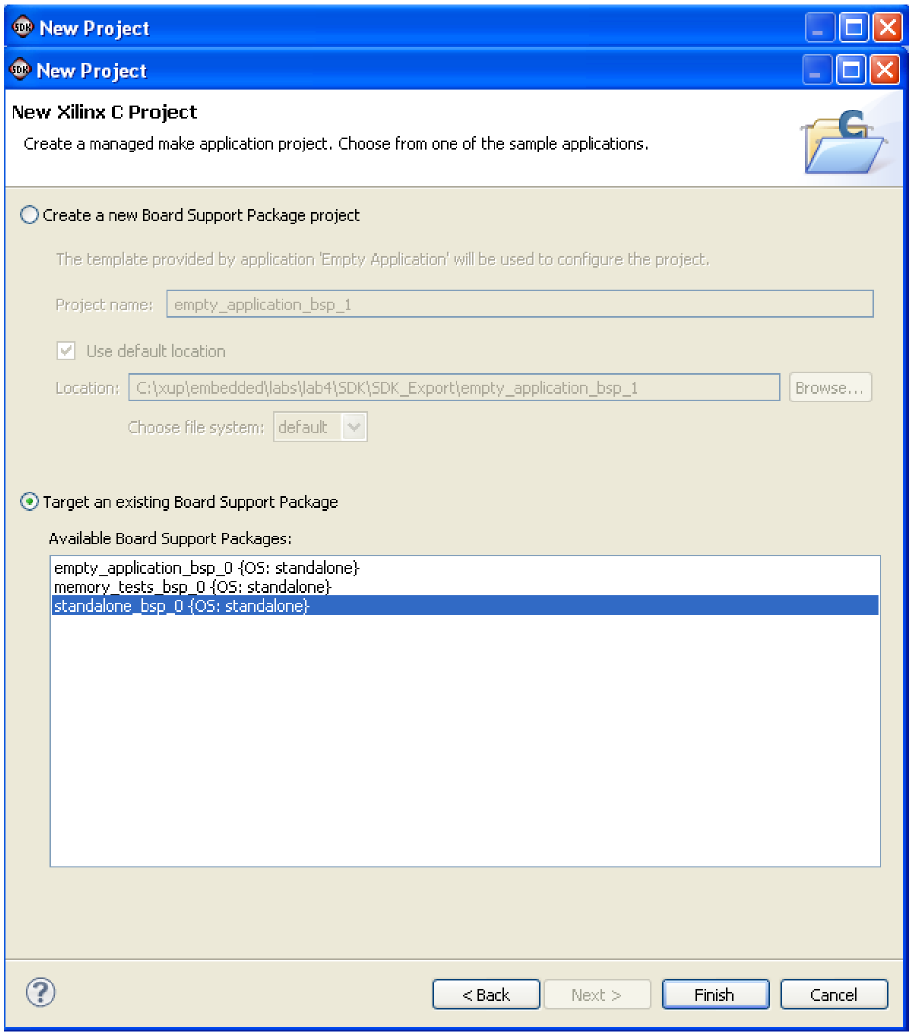
Dhrystone
Empty Application
Hello World
lwIP Echo Server
Memory Tests
Peripheral Tests
SREC Bootloader
Xilkernel POSIX Threads Demo

Description

A blank C project.

Figure 5. Create a blank C Project

2-1-8. Select **Target** an existing **Board Support Package** and click **Finish**.


New Project
New Project
New Xilinx C Project
Create a managed make application project. Choose from one of the sample applications.

Create a new Board Support Package project

The template provided by application 'Empty Application' will be used to configure the project.

Project name: empty_application_bsp_1

Use default location [✓]

Location: C:\xup\embedded\labs\lab4\SDK\SDK_Export\empty_application_bsp_1

Choose file system: default

[✓ Target an existing Board Support Package]

Available Board Support Packages:

empty_application_bsp_0 {OS: standalone}
memory_tests_bsp_0 {OS: standalone}
[✓ standalone_bsp_0 {OS: standalone}]

Figure 6. Use Existing Board Support Package

The TestApp project will be created in the Project Explorer window of SDK.

2-2. Import lab4.c file

2-2-1. Select **TestApp** in the project view, right-click, and select **Import.**

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

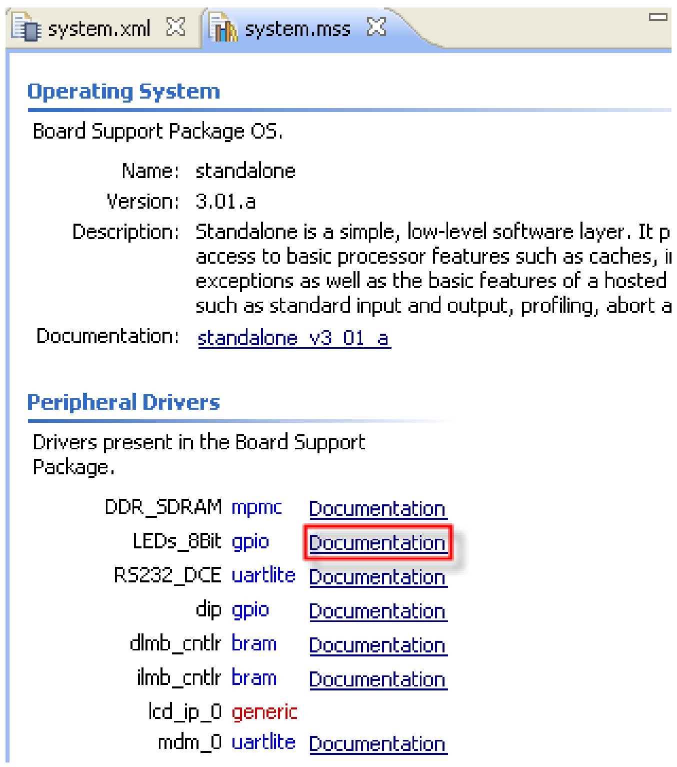
2-2-3. Browse to **W:\A.Bukowiec\zajecia\SW\source** folder.

2-2-4. Select **lab4.c** and click **Finish.**

This will compile the source file and generate **TestApp.elf** in the **D:\lab4\TestApp\Debug** folder

You will extend the functionality in **lab4.c** by adding code to display switch settings on the LEDs.

2-2-5. Open the GPIO API documentation by clicking on **Documentation** link of **LEDs_8Bit** peripheral under the Peripheral Drivers section to opne the documentation in a default browser window.



Operating System

Board Support Package OS.

Name: standalone
Version: 3.01.a
Description: Standalone is a simple, low-level software layer. It p
access to basic processor features such as caches, i
exceptions as well as the basic features of a hosted
such as standard input and output, profiling, abort a
Documentation: standalone_v3_01_a

Peripheral Drivers

Drivers present in the Board Support Package.

DDR_SDRAM	mpmc	Documentation
LEDs_8Bit	gpio	Documentation
RS232_DCE	uartlite	Documentation
dip	gpio	Documentation
dlmb_cntlr	bram	Documentation
ilmb_cntlr	bram	Documentation
lcd_ip_0	generic	
mdm_0	uartlite	Documentation

Figure 7. Accessing Device Driver Documentation

2-2-6. View the various C and Header files associated with the GPIO by selecting **File List** at the top.

2-2-7. Click the header file **xgpio.h** and review the list of available function calls for the GPIO.

2-2-8. The following steps must be performed in the software application to enable writing to the GPIO: **1) Initialize the GPIO, 2) Set data direction, and 3) Write 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 discretes are input and which are output. Bits set to 0 are output and bits set to 1 are input.

XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned channel,u32 data)

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

2-2-9. 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.

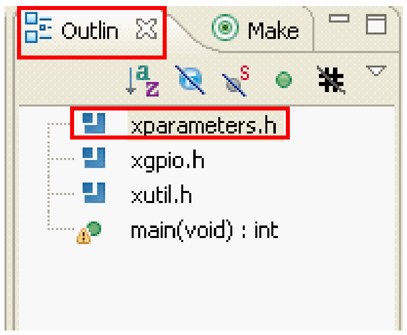


Figure 8. Double-Click the Generated Header File

- In the xparameters.h file, find the following #define used to identify the LEDs_8Bit peripheral:

#define XPAR_LEDS_8BIT_DEVICE_ID 1 ⟵ **Note: The number might be different**

Note: The LEDS_8BIT matches the instance name assigned in the MHS file for this peripheral.

This #define can be used in the XGpio_Intialize function call.

Task 1

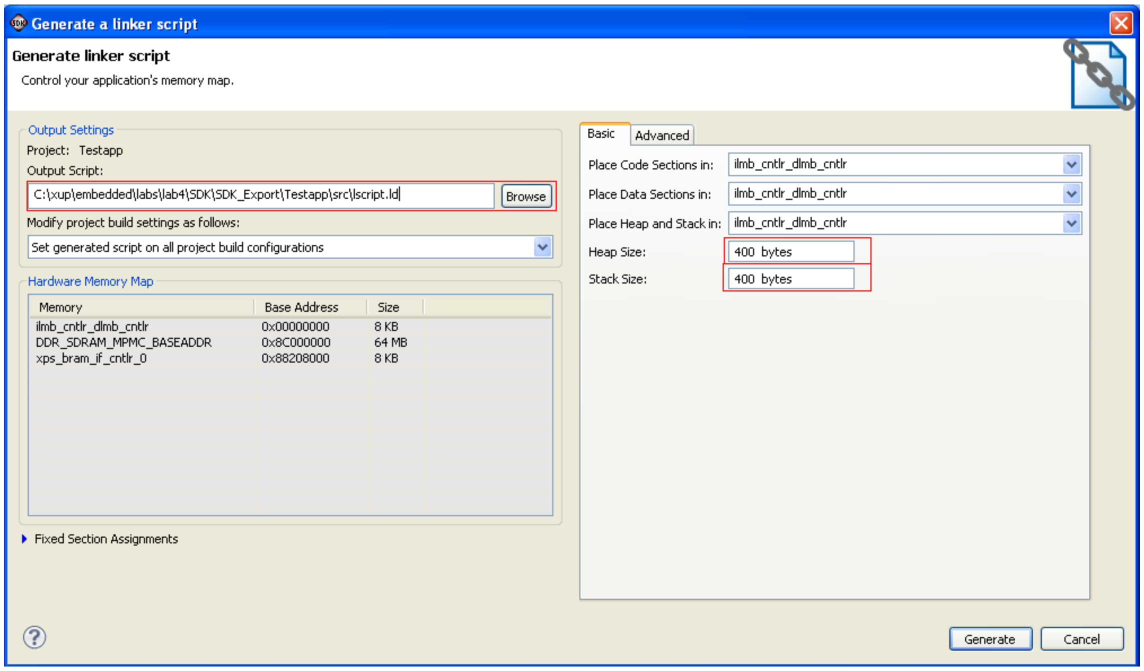
Modify the C code to echo DIP Switch settings also on the LEDs.

Analyze Assembled Object Files Step 3

3-1. Generate linker script targeting .text section to ilmb and setting heap and stack sizes to 400 each. Compile the application, and analyze the assembled object files using the objdump utility.

3-1-1. Select **Xilinx Tools → Generate Linker Script**...

3-1-2. Select **lscript.ld** under **D:\lab4\SDK\SDK_Export\Testapp\src** and click **Save**. Change Heap and Stack sizes to 400 each (to fit the program into single memory) followed by clicking **Generate.**



Project: Testapp
Output Script: C:\xup\embedded\labs\lab4\SDK\SDK_Export\Testapp\src\lscript.ld
Modify project build settings as follows: Set generated script on all project build configurations

Memory	Base Address	Size
ilmb_cntlr_dlmb_cntlr	0x00000000	8 KB
DDR_SDRAM_MPMC_BASEADDR	0x8C000000	64 MB
xps_bram_if_cntlr_0	0x88208000	8 KB

Place Code Sections in: ilmb_cntlr_dlmb_cntlr
Place Data Sections in: ilmb_cntlr_dlmb_cntlr
Place Heap and Stack in: ilmb_cntlr_dlmb_cntlr
Heap Size: 400 bytes
Stack Size: 400 bytes

Figure 10. The Linker Script GUI

3-1-3. Click **Yes** to overwrite the linker script.

3-1-4. Click somewhere in white space area in the **lab4.c** file, add a space and save it to recompile the program.

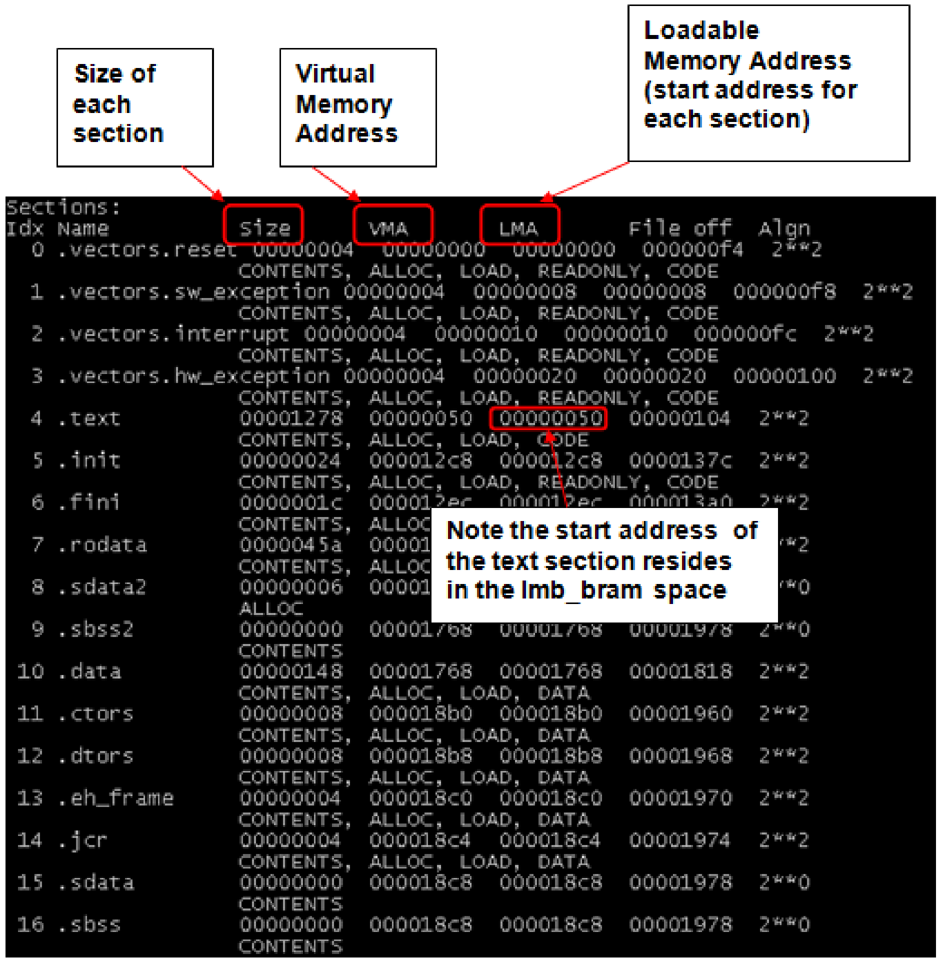
3-1-5. Launch the Bash shell by selecting **Xilinx Tools → Launch shell.**

3-1-6. Change the directory to ***TestApp/Debug*** using the **cd** command in the bash shell.

You can determine your directory path by using the **pwd** command.

3-1-7. Type ***mb-objdump -h TestApp.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:


Size of each section
Virtual Memory Address
Loadable Memory Address (start address for each section)
Note the start address of the text section resides in the lmb_bram space

```
Sections:
Idx Name              Size      VMA       LMA       File off  Algn
  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             00001278  00000050  00000050  00000104  2**2
                      CONTENTS, ALLOC, LOAD, CODE
  5 .init             00000024  000012c8  000012c8  0000137c  2**2
                      CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .fini             0000001c  000012ec  000012ec  000013a0  2**2
                      CONTENTS, ALLOC
  7 .rodata           0000045a  00001                         *2
                      CONTENTS, ALLOC
  8 .sdata2           00000006  00001                         *0
                      ALLOC
  9 .sbss2            00000000  00001768  00001768  00001978  2**0
                      CONTENTS
 10 .data             00000148  00001768  00001768  00001818  2**2
                      CONTENTS, ALLOC, LOAD, DATA
 11 .ctors            00000008  000018b0  000018b0  00001960  2**2
                      CONTENTS, ALLOC, LOAD, DATA
 12 .dtors            00000008  000018b8  000018b8  00001968  2**2
                      CONTENTS, ALLOC, LOAD, DATA
 13 .eh_frame         00000004  000018c0  000018c0  00001970  2**2
                      CONTENTS, ALLOC, LOAD, DATA
 14 .jcr              00000004  000018c4  000018c4  00001974  2**2
                      CONTENTS, ALLOC, LOAD, DATA
 15 .sdata            00000000  000018c8  000018c8  00001978  2**0
                      CONTENTS
 16 .sbss             00000000  000018c8  000018c8  00001978  2**0
                      CONTENTS
```

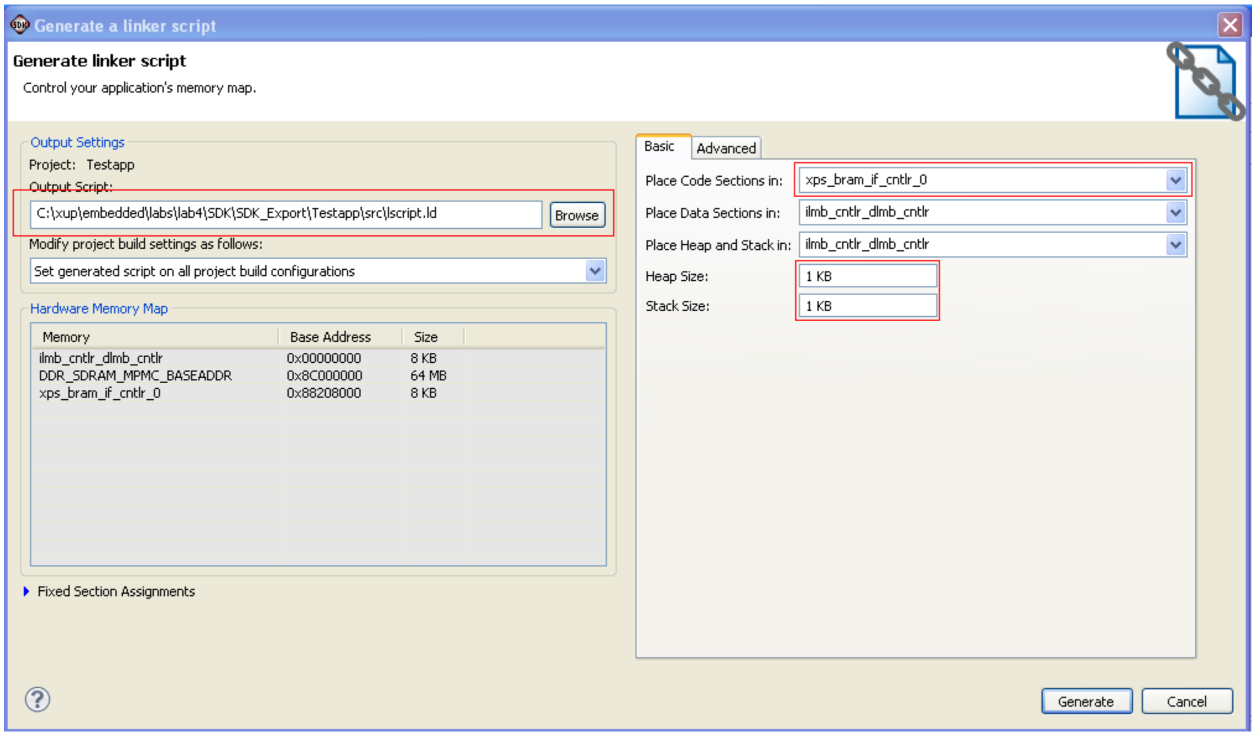
Figure 11. Object Dump Results -.text in ILMB space

3-2. Change the location of the text section so that it resides in the XPS PLB memory. Recompile the code, re-execute the objdump command, and analyze the output.

3-2-1. Select **Xilinx Tools → Generate Linker Script...**

3-2-2. Select **lscript.ld** under **D:\lab4\SDK\SDK_Export\Testapp\src** and click **Save**. Notice that the Heap and Stack sizes are changed to 1 KB (it will be OK as we are going to target the .text section to XPS BRAM).

3-2-3. Select the Code section to target in to XPS BRAM and click **Generate.**


Generate a linker script

Generate linker script

Control your application's memory map.

Output Settings

Project: Testapp

Output Script:

C:\xup\embedded\labs\lab4\SDK\SDK_Export\Testapp\src\lscript.ld

Modify project build settings as follows:

Set generated script on all project build configurations

Hardware Memory Map

Memory	Base Address	Size
ilmb_cntlr_dlmb_cntlr	0x00000000	8 KB
DDR_SDRAM_MPMC_BASEADDR	0x8C000000	64 MB
xps_bram_if_cntlr_0	0x88208000	8 KB

Fixed Section Assignments

Basic | Advanced

Place Code Sections in: xps_bram_if_cntlr_0

Place Data Sections in: ilmb_cntlr_dlmb_cntlr

Place Heap and Stack in: ilmb_cntlr_dlmb_cntlr

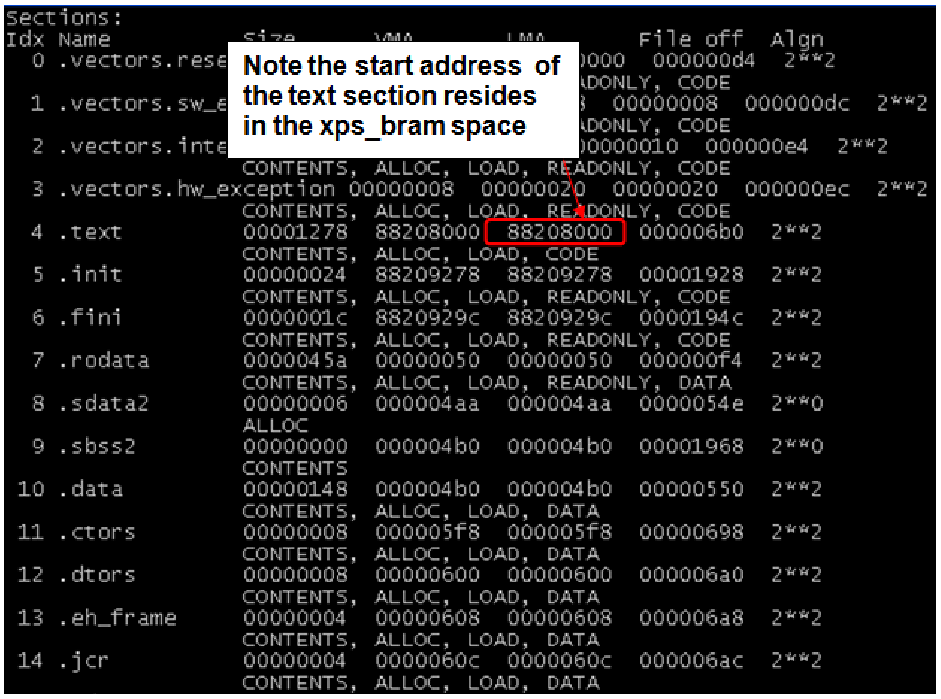
Heap Size: 1 KB

Stack Size: 1 KB

Figure 12. Target .text Section to XPS BRAM

3-2-4. Click somewhere in white space area in the **lab4.c** file, add a space and save it to recompile the program.

3-2-5. Execute the ***mb-objdump -h TestApp.elf*** command in the bash shell.


```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .vectors.rese                              0000  000000d4  2**2
                                               ADONLY, CODE
  1 .vectors.sw_e                              8  00000008  000000dc  2**2
                                               ADONLY, CODE
  2 .vectors.inte                              00000010  000000e4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .vectors.hw_exception 00000008  00000020  00000020  000000ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .text         00001278  88208000  88208000  000006b0  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  5 .init         00000024  88209278  88209278  00001928  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .fini         0000001c  8820929c  8820929c  0000194c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 .rodata       0000045a  00000050  00000050  000000f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .sdata2       00000006  000004aa  000004aa  0000054e  2**0
                  ALLOC
  9 .sbss2        00000000  000004b0  000004b0  00001968  2**0
                  CONTENTS
 10 .data         00000148  000004b0  000004b0  00000550  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 11 .ctors        00000008  000005f8  000005f8  00000698  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 .dtors        00000008  00000600  00000600  000006a0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 13 .eh_frame     00000004  00000608  00000608  000006a8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 14 .jcr          00000004  0000060c  0000060c  000006ac  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Note the start address of the text section resides in the xps_bram space

Figure 13. Object Dump Results -.text in XPS BRAM Space

Verify in Hardware Step 4

4-1. Connect and power the board. Select TestApp.elf as the application to intialize the BRAM with and program the FPGA from SDK.

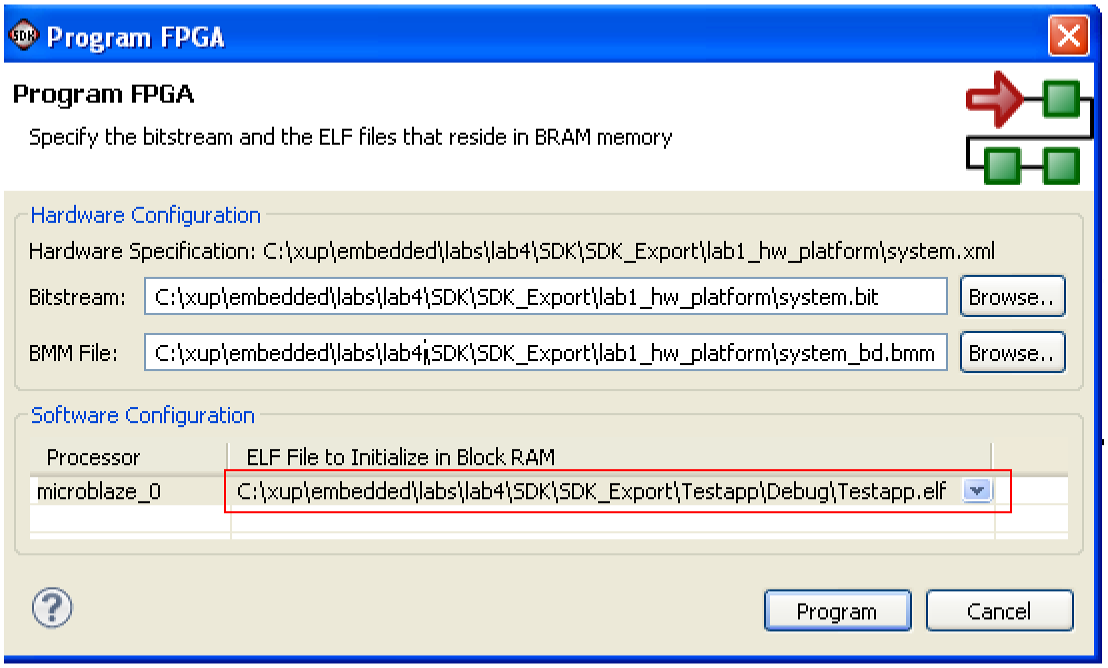
4-1-1. Connect and power the board

4-1-2. Open a PuTTY with the following settings

- Baud rate: 115200
- Data bits: 8
- Parity: none
- Stop bits: 1
- Flow control: none

4-1-3. In SDK, select **Xilinx Tools → Program FPGA.**

4-1-4. Click on the drop-down button of the Software Configuration and select **TestApp.elf** application to be targeted in BRAM.


Program FPGA

Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Specification: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system.xml

Bitstream: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system.bit

BMM File: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system_bd.bmm

Software Configuration

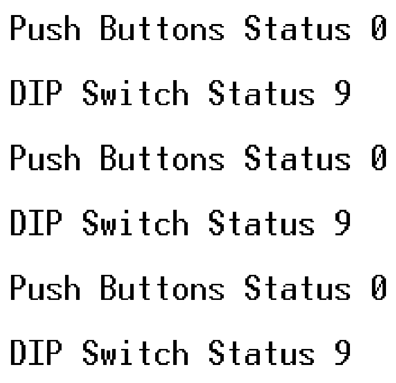
Processor	ELF File to Initialize in Block RAM
microblaze_0	C:\xup\embedded\labs\lab4\SDK\SDK_Export\Testapp\Debug\Testapp.elf

Figure 14. Assigning TestApp to BRAM Space

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

4-1-6. Flip the DIP switches and verify that the LEDs will light according to the switch settings.

Verify that you see the results of the DIP switch and Push button settings in Hyper Terminal.



```
Push Buttons Status 0
DIP Switch Status 9
Push Buttons Status 0
DIP Switch Status 9
Push Buttons Status 0
DIP Switch Status 9
```

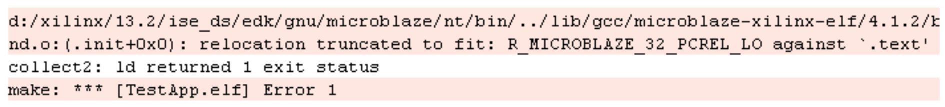
Figure 15. DIP Switch and Push Button settings displayed in PuTTY

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

4-2. Change one of the xil_printf function calls to printf. Re-compile the code and observe that the XPS BRAM space is not sufficient. Generate the linker script to target the .text section to external memory (DDR2_SDRAM).

4-2-1. In the text editor, change the **xil_printf** function call to **printf.**

4-2-2. Compile the code and observe the output in the console window.



```
d:/xilinx/13.2/ise_ds/edk/gnu/microblaze/nt/bin/../lib/gcc/microblaze-xilinx-elf/4.1.2/b
nd.o:(.init+0x0): relocation truncated to fit: R_MICROBLAZE_32_PCREL_LO against `.text'
collect2: ld returned 1 exit status
make: *** [TestApp.elf] Error 1
```

Figure 16. Errors Shown in Console Window

Observe in the console window that the .text section is too big and sections overlap.

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

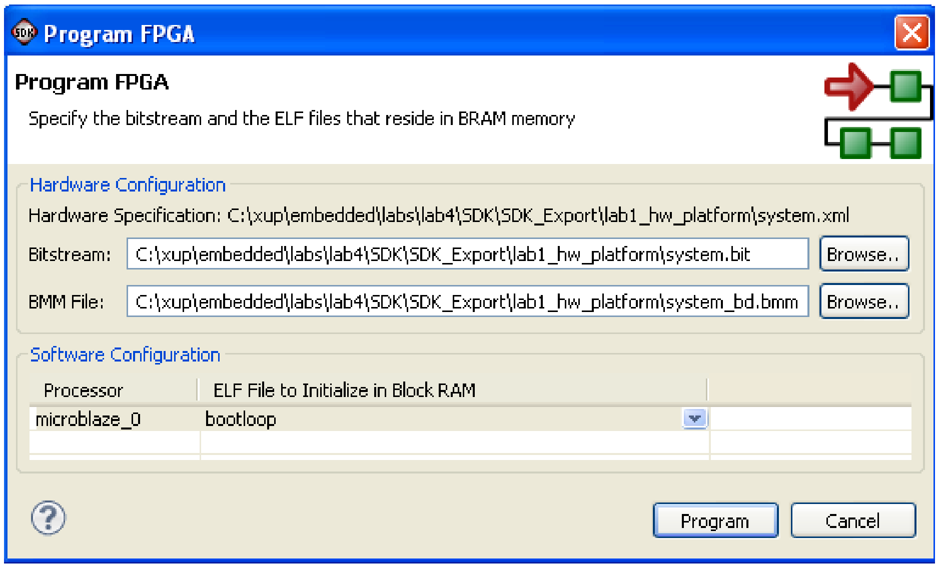
4-2-4. Target **.text** section to **DDR_SDRAM** memory, click **Generate**, and click **Yes** to overwrite.

4-2-5. Click somewhere in white space area in the **lab4.c** file, add a space and save it to recompile the program.

4-3. Initialize BRAM with bootloop application. Program the FPGA. Start the xmd console window from SDK. Download the TestApp.elf application from the xmd console window, and run the application.

As the .text section is targeted to external memory, the FPGA must be initialized with a bootloop program, and the test application must be downloaded and executed from XMD shell.

4-3-1. In SDK, select **Xilinx Tools → Program FPGA**, Click **Program** to initialized the BRAM with the bootloop program, and program the FPGA.



Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Specification: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system.xml

Bitstream: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system.bit

BMM File: C:\xup\embedded\labs\lab4\SDK\SDK_Export\lab1_hw_platform\system_bd.bmm

Software Configuration

Processor	ELF File to Initialize in Block RAM
microblaze_0	bootloop

Figure 17. Initialize BRAM with Bootloop and Program the FPGA

4-3-2. In SDK, select **Xilinx Tools → XMD Console** to open an XMD console window.

4-3-3. In the XMD console window, type **cd D:\lab4\SDK\SDK_Export\Testapp\Debug**.

4-3-4. In the XMD console window, type **connect mb mdm.**

4-3-5. In the XMD console window, type **dow Testapp.elf.**

This will download the application in the external memory.

4-3-6. In the XMD window, type **con** to execute the program.

Observe the Hyper Terminal window as the program executes. Play with dip switches and observe the LEDs

4-3-7. In the XMD console window, type **stop** to stop the program execution. Close SDK and XPS.

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 MSS file created in XPS and let you update the settings so you can represent 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. When the application is too big to fit in the internal BRAM, you can download the application in external memory using XMD, and then execute the program.