

***Lab 14***  
***Writing Basic Software Applications***  
***Lab: MicroBlaze***

# Writing Basic Software Applications Lab: MicroBlaze

---

## Introduction

---

This lab guides you through the process of writing a basic software application. The software will write to one of the OPB GPIOs; LEDs. Xilinx Platform Studio (XPS) will write the code and create an MSS file for LibGen.

---

## Objectives

---

After completing this lab, you will be able to:

- Write a basic application to access an IP peripheral
- Utilize XPS to generate a MSS file
- Generate a bit file
- Download the bit file and verify in hardware (if hardware is available)
- Develop a simple linker script

---

## Procedure

---

The first three labs defined the hardware for the processor system. This lab comprises several steps, including the writing of a basic software application to access one of the peripherals specified in Lab2mb. Below each general instruction for a given procedure, you will find accompanying step-by-step directions and illustrated figures providing more detail for performing the general instruction. If you feel confident about a specific instruction, feel free to skip the step-by-step directions and move on to the next general instruction in the procedure.

**Note:** If you are unable to complete the lab at this time, you can download the lab files for this module from the Xilinx University Program site at <http://university.xilinx.com>

---

## Opening the Project

## Step 1

---



- ❶ Create a **lab14mb** folder in the **c:\xup\embedded\labs** directory. If you wish to continue with your completed design from lab3 then copy the contents of the **lab3mb** folder into the **lab4mb** folder, otherwise, if you wish to start with a known good design, then copy the contents of **c:\xup\embedded\mb\_completed\lab3mb** into the **lab4mb** directory.
- ❷ Open XPS by clicking **Start → Programs → Xilinx Platform Studio → Xilinx Platform Studio**
- ❸ Click **File → Open Project** and browse to the project which in the directory:  
**c:\xup\embedded\labs\lab14mb**
- ❹ Click **system.xmp** to open the project

---

## Creating a BSP

## Step 2

---



Specify the MicroBlaze™ processor standalone operating system and driver interface level.

The BSP is created based upon the peripherals included in the design.

- ❶ Double-click **microblaze\_0** from the **System BSP** hierarchy, as shown in Figure 10c-1. You can also open the same dialog box by clicking **Project → Software Platform Settings**

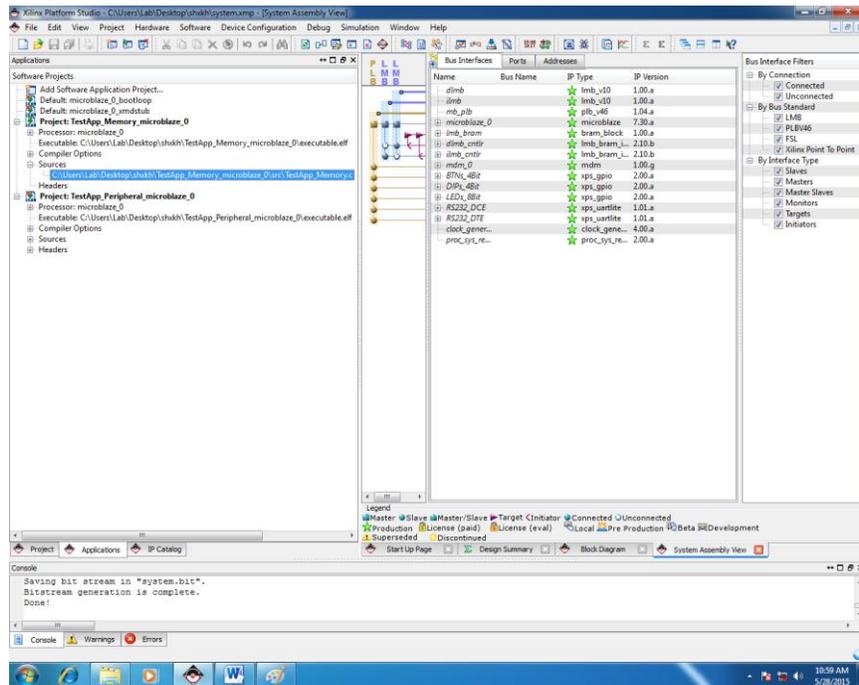
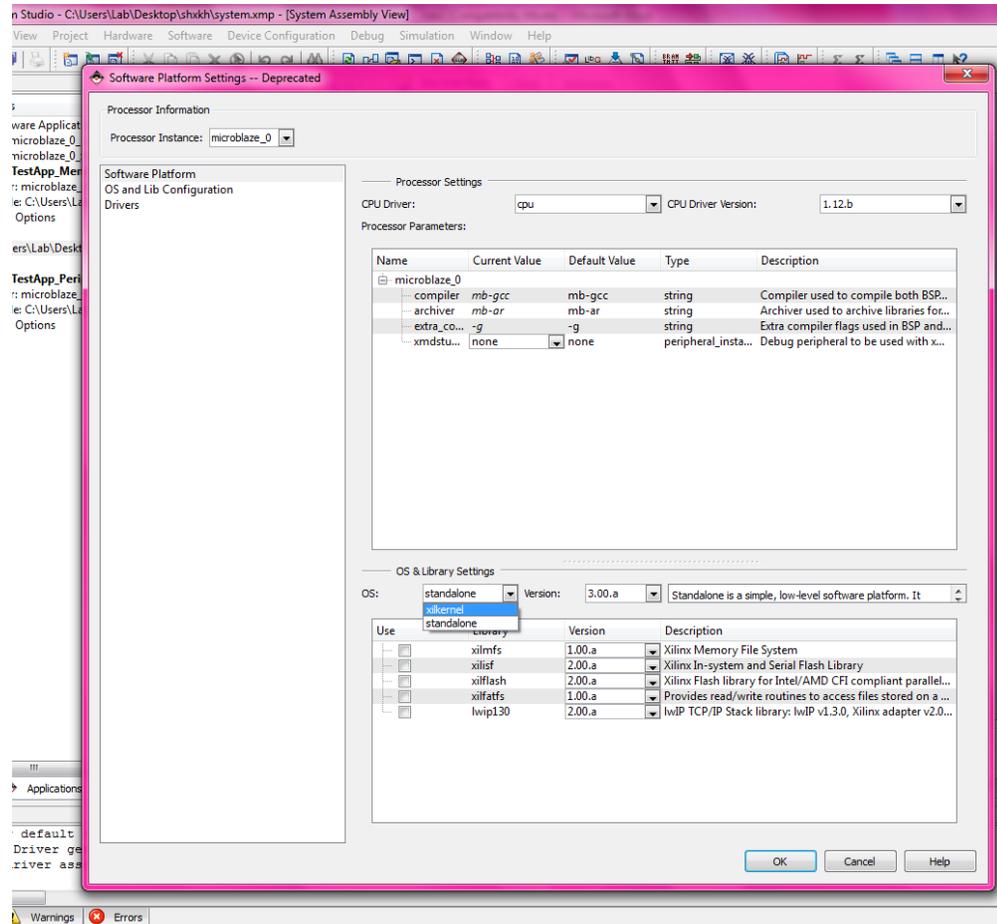


Figure 10c-1. Opening S/W Settings for the microblaze\_0 Instance

This will open the **Software Platform Settings** dialog with the **Software Platform** tab.



**Figure 10c-2. Software Platform Settings for the microblaze\_0 Instance**

- ③ In the **Software Platform** tab, the **Driver** can be selected for each of the peripherals in the system. You can also select the **Kernel and Operating Systems** for each of the processor instances. In addition, supporting libraries can be selected if they will be used. Make sure that the settings are as displayed in Figure 10c-2

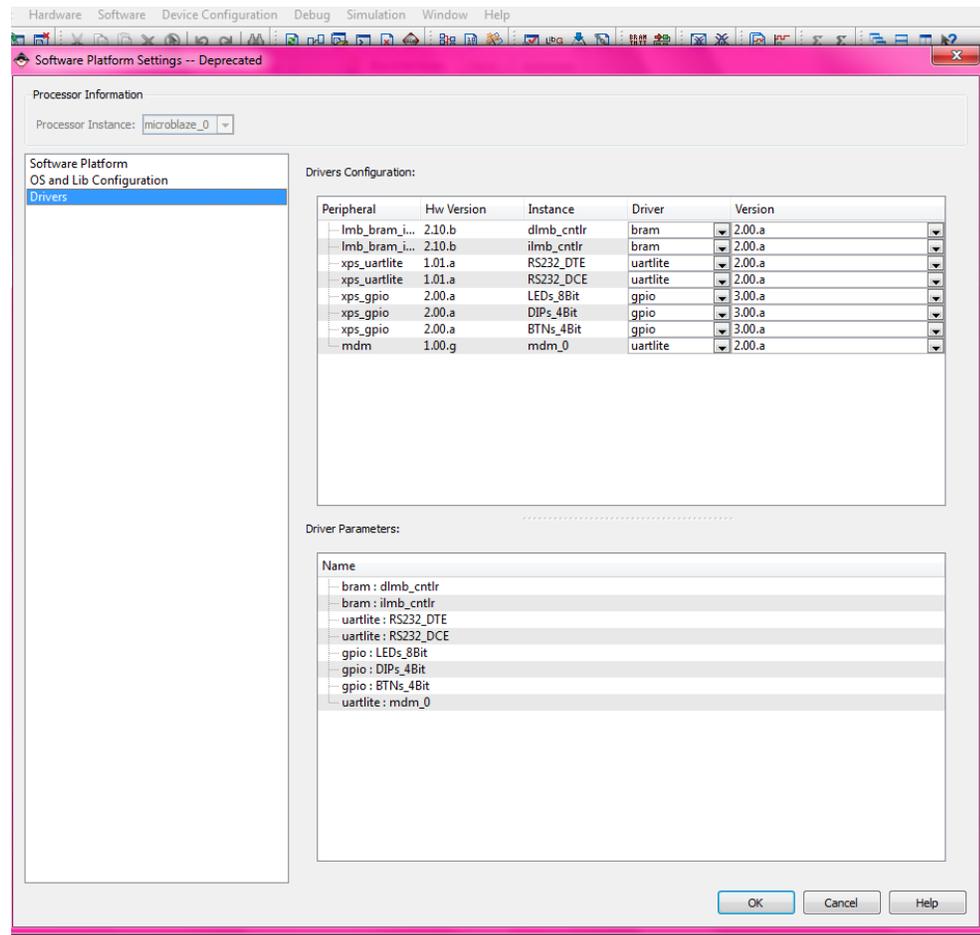
- ③ Click on the **Processor and Driver Parameters** tab and specify the following parameters:

**Processor Parameters: Instance**

compiler – mb-gcc  
 archiver – mb-ar  
 EXTRA\_COMPILER\_FLAGS – -g  
 xmdstub\_peripheral – none  
 CORE\_CLOCK\_FREQ\_HZ – 50000000

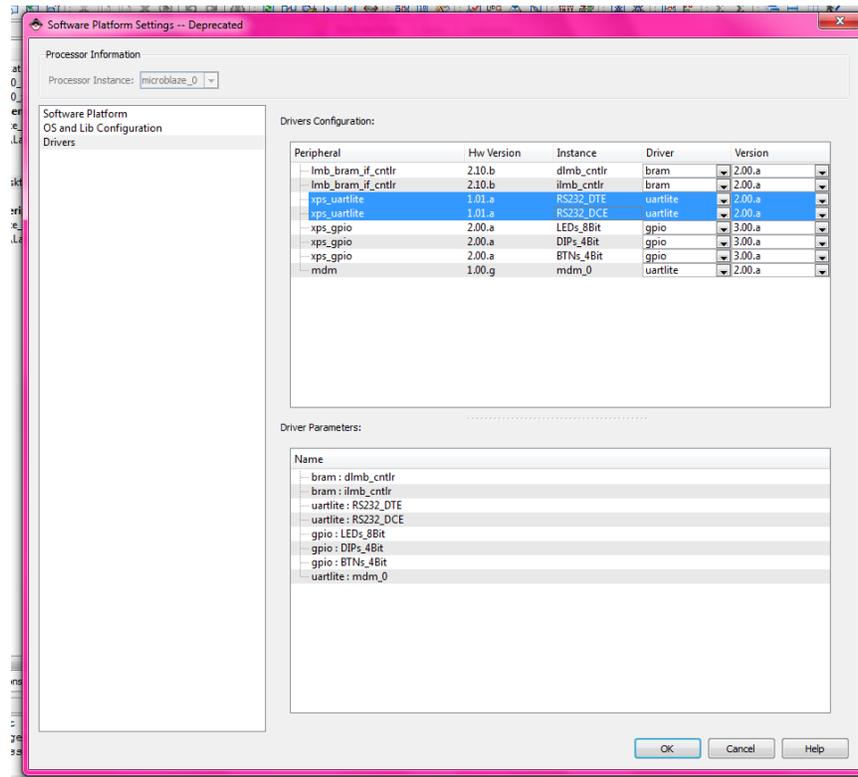
**Driver Parameters: Instance**

Leave Blank



**Figure 10c-3. Processor and Driver Parameters Tab of the Software Platform Settings for the microblaze\_0 Instance**

- ④ Click the Library/OS Parameters tab



**Figure 10c-4. Library/OS Parameters Tab of the Software Platform Settings for the microblaze\_0 Instance**

- 5 Click the **Current Value** field for **stdin** and select **RS232**. Similarly, click the **Current Value** field for **stdout** and select **RS232**. This will assign the **uart** device as the **stdin** and **stdout**. Base System Builder already set these when the RS232 peripheral was included. If a system does not have any stdin/stdout devices, then keep the current value as **None**. Leave the **Current Value** for **need\_xil\_malloc** as false because your application does not use any malloc function call
- 6 Click **OK** to accept the settings



Generate the BSP.

- 1 Double-click the **system.mss** file under the **System** tab in XPS, as shown in Figure 10c-5  
This will open the MSS file for this project.

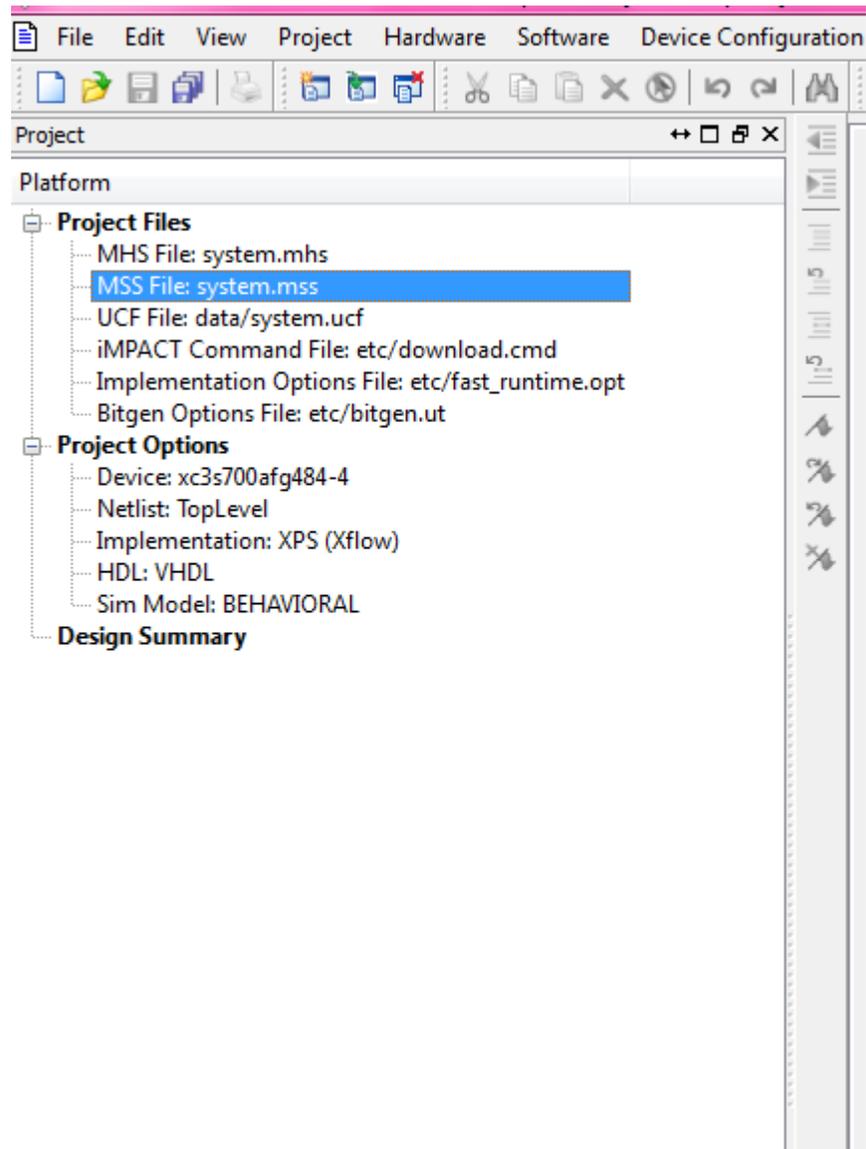


Figure 10c-5. System Tab

- ③ XPS has written the parameters that are specified in the **Peripheral Options** to the **system.mss** file



1. List the assigned driver to each of the following peripheral instances:

mb_opb:	_____
debug_module:	_____
dlmb_cntlr:	_____
ilmb_cntlr:	_____
LEDs_8Bit:	_____
Push_Buttons_3Bit:	_____
DIP_Switches_8Bit:	_____
delay:	_____
RS232:	_____
opb_7segled_0:	_____



2. Why do some of the above mentioned devices not have a specific driver?

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

- ③ Close the **system.mss** file

- ④ Generate the **BSP** by clicking **Tools → Generate Libraries** or click the  button

This will run LibGen on the system.mss file to generate the BSP library files.



3. List the created subfolders, their contents, and their possible purposes under the **microblaze\_0** folder in the lab4mb folder.

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## Creating a Basic C File

## Step 3



Add a software project and create a basic C file to write data to the **LEDs\_8Bit** peripheral.

- 1 In the **Applications** Tab double Click on **Software Projects**

This will open a dialog box to create a new project

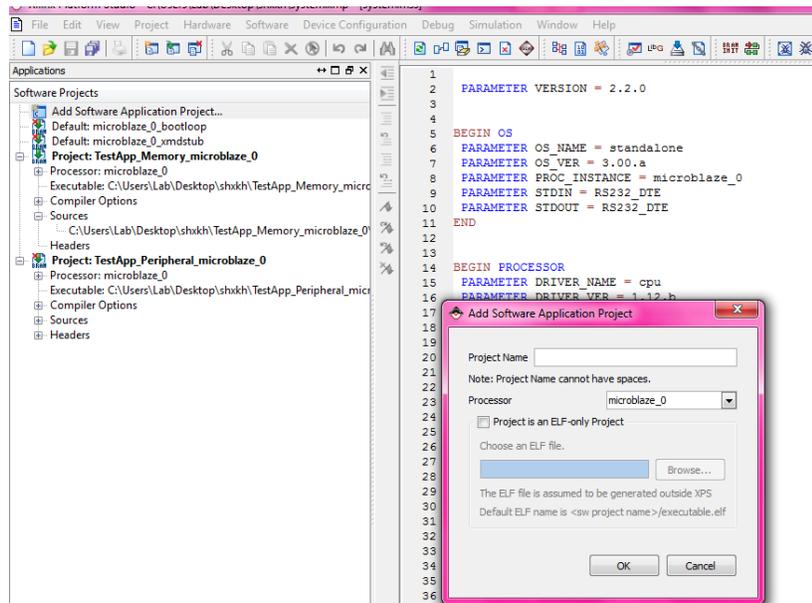
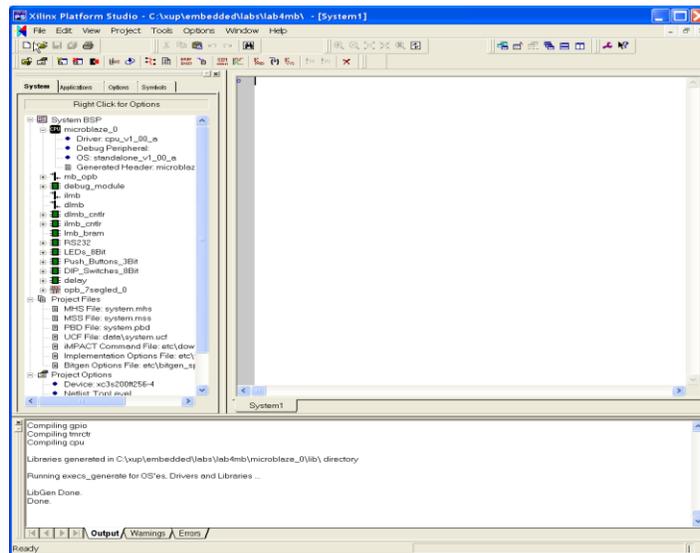


Figure 10c-6. New Project

- 2 Enter the name **MyProj** and click **OK**.
- 3 Click **File** → **New**

This will open a dialog box to create a new project

This will open a new document in the XPS editor



**Figure 10c-7. New Document in the XPS Editor**

The first step is to add the header files for the required functions. All of the header files related to this project were placed in the `microblaze_0/include` directory when LibGen was run.

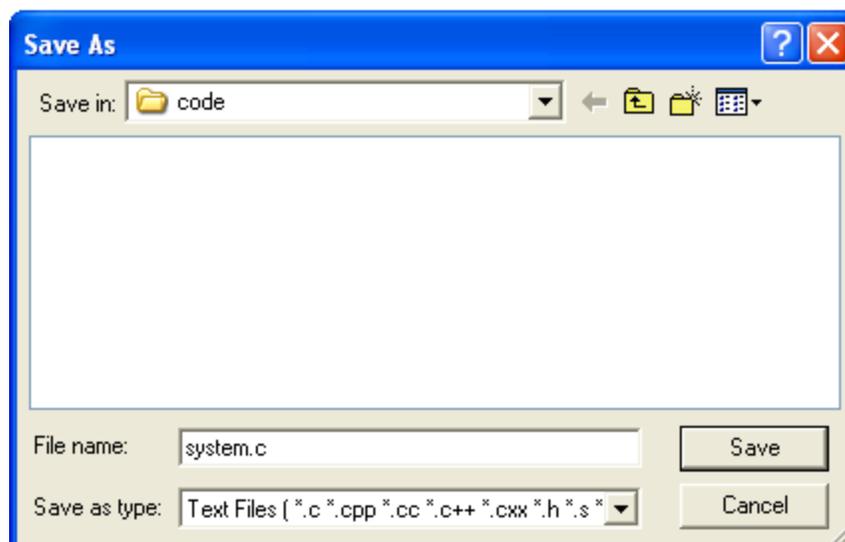
④ Add the following to the C file:

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

⑤ Click File → Save As

This will open the Save As dialog.

⑥ Create a new directory named **code** in the **lab4mb** directory and save the file as **system.c**, as shown in Figure 10c-8



**Figure 10c-8. Save As Dialog**

- 7 Add the following to the C file

```
main() {
```

The first step in main is to initialize the **GPIO peripheral**



Write a **main()** function that will initialize and set the data direction for the **LEDs\_8Bit** peripheral to output.

- 1 From the Windows start menu Click **Start** → **Programs** → **Xilinx Platform Studio 6.2i** → **EDK 6.2 Documentation**
- 2 Click the **Documents** link
- 3 Scroll down and click **Driver Reference Guide**

This will open the **xilinx\_drivers.pdf** file.

- 4 In Acrobat Reader, click  and search for the **XGpio\_Initialize** function. You may have to click **Find Next** to observe the function description page
- 5 This documentation contains a detailed description of the **XGpio\_Initialize** function
  - The documentation outlines two parameters that **XGpio\_Initialize** requires:
  - *InstancePtr* is a pointer to an XGpio instance. The memory that the pointer references must be preallocated 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.

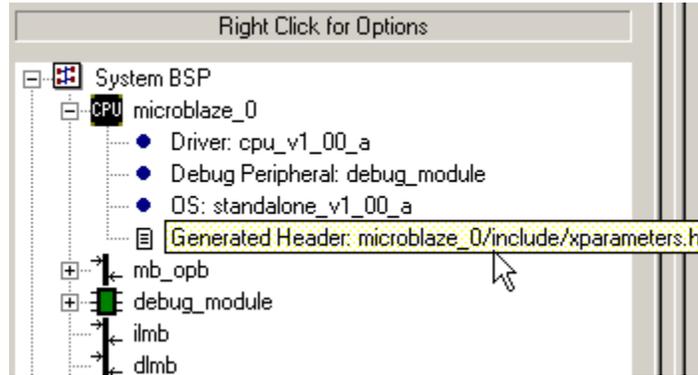
- 6 Define an **XGpio** type variable named **gp\_out**. This variable will be used as the first parameter in the **Xgpio\_Initialize** function call

- 7 Add the variable to the function call. It should now look like the following:

```
XGpio_Initialize(&gp_out,
```

The second parameter is the device ID for the device that you want to initialize. This information can be found in the **xparameters.h** file.

- 8 Under the System BSP microblaze\_0 instance, double-click the **Generated Header: microblaze\_0/include/xparameters.h** entry, as shown in Figure 10c-9



**Figure 10c-9. Double-click the Generated Header File**

- LibGen writes the **xparameters.h** file, and the file provides critical information for driver function calls.
- This function call initializes the GPIO that is used as an output for the LEDs found on the board. In the *xparameters.h* file, find the following #define used to identify the LEDS\_8BIT peripheral:

*#define XPAR\_LEDS\_8BIT\_DEVICE\_ID 0* ● — **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 as the device ID in the function call.

- ⑨ Add the **device ID** to the **function call** so that it looks like the following:

```
XGpio_Initialize(&gp_out, XPAR_LEDS_4BIT_DEVICE_ID);
```

The file should now look like Figure 10c-10.

```
00 #include "xparameters.h"
01 #include "xgpio.h"
02
03 main()
04 {
05     XGpio gp_out;
06
07     XGpio_Initialize(&gp_out, XPAR_LEDS_8BIT_DEVICE_ID);
```

**Figure 10c-10. Partially Completed C File**

- ⑩ Refer to the documentation to determine how to set all of the bits of the **GPIO bus as outputs**. This will involve using the **XGpio\_SetDataDirection** function call. Add code to perform this function, and then save the file.

```
XGpio_SetDataDirection (&gp_out, 1, 0x00);
```



Write a counter that continuously counts from 0 to 255 to drive the **LEDs\_8Bit** peripheral output. Output the current value of the counter by using the appropriate function. Write a software pause for loop to pause the output between each count.

- ① Write code to implement a counter that continuously counts from 0 to 255. The count will be used to drive the output of the LEDs\_8Bit peripheral

The following code is provided as an example:

```
while(1) {
    j = (j+1) % 256; }
```

You will also need to include the declaration of the variable *j*.

- ② Output the current value of *j* to the **LEDs\_8Bit**



4. Using the information in the **gpio.h** documentation (the gpio information in the PDF file that is open), which function can be used?

- ③ Add this function and the appropriate parameters to the C code

**Note:** Because the MicroBlaze™ processor does not have an internal timer or counter, a delay loop must be created in the software.

- ④ Add the following code to create a software delay to pause between each count that gets displayed:

```
int i;  
for(i=0; i<80000; i++);
```

- ⑤ The final C program should look like Figure 10c-11

```
00 #include "xparameters.h"  
01 #include "xgpio.h"  
02  
03 main()  
04 {  
05 XGpio gp_out;  
06 int i=0;  
07 int j=0;  
08  
09 XGpio_Initialize (&gp_out, XPAR_LEDS_8BIT_DEVICE_ID);  
10 XGpio_SetDataDirection (&gp_out, 1, 0x00);  
11  
12 while (1)  
13 {  
14     j = (j + 1) % 256;  
15  
16     //write the value of j to the LEDs  
17     XGpio_DiscreteWrite (&gp_out, 1, j);  
18  
19     //software delay loop for pause  
20     for (i=0; i<10; i++);  
21 }  
22 }  
23  
24
```

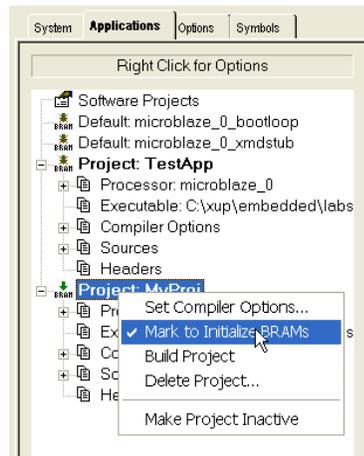
**Figure 10c-11. Final C Program**

- ⑥ Save and close the file



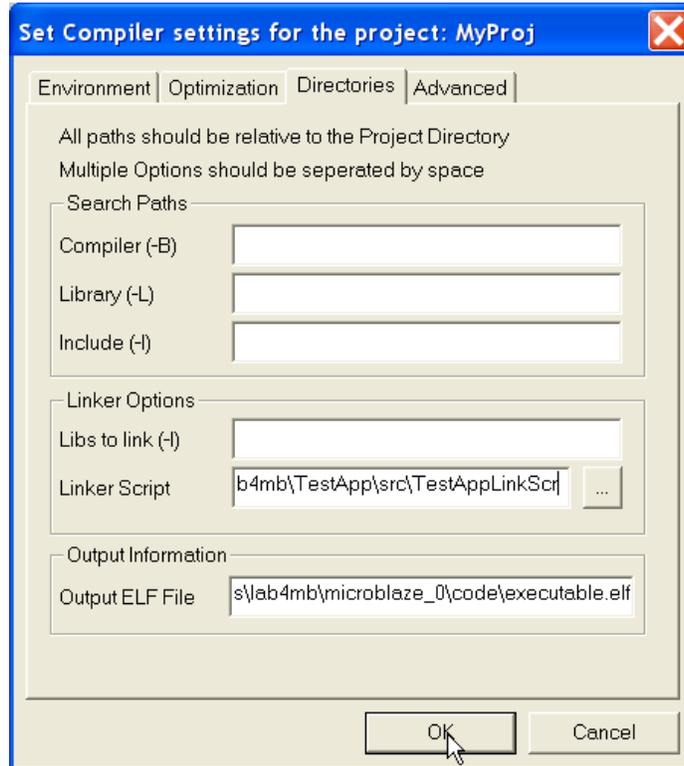
Select the **Application** Tab, add the source code (system.c from the code directory) and change the SW project TestApp so that **TestApp.c** is no longer initialized in BRAM. Compile the source code.

- ① Click the **Application** tab to view the current project's **Compiler Options** and **Sources**
- Right-click **Project: TestApp** and unselect **Mark to Initialize BRAMs**
  - Verify that the small green arrow next to **Project: MyProj** does *not* have a red x through it. If it does, Right-click **Project: MyProj** and toggle **Mark to Initialize BRAMs**.



**Figure 10c-12. Applications Tab for the Current Project**

- ③ Right-click **Sources** under **Project: MyProj** and select **Add File**
- ③ Browse to the **code** directory under the current project (lab4mb) and select the **system.c** file
- ④ Double-click **Compiler Options** under **Project: MyProj** in the Application tab
- ⑤ Click the **Directories** tab to set the directory options.
  - Change the **Output ELF File** entry to place the **executable.elf** file in the **microblaze\_0\code** directory under the current project directory.
  - For **Linker Script** Browse to the **TestAppLinkScr** file in the **TestApp/Src** directory.
  - The directory options should be set as shown in Figure 10c-13



**Figure 10c-13. Assigning the Destination Directory for the Executable File**

- ⑥ Click **OK** to accept the setting
- ⑦ Compile the C code by clicking the  button

 5. After the program has compiled determine the sizes of the program sections:

.text section (code):	
.data section (variables):	
.bss section (heap and stack):	
Total size in decimal:	
Total size in hexadecimal:	

## Linker Script

## Step 4



Compile the C code by clicking the  button. Change the directory to `c:/xup/embedded/labs/lab4mb/microblaze_0/code`. Execute the `mb-objdump -h executable.elf` command in the **Xygwin** shell and analyze the output.

- ❶ Start a Xygwin shell by clicking the  button. This should place you in the project directory.
- ❷ Change the directory to `c:/xup/embedded/labs/lab4mb/microblaze_0/code` by using the `cd` command
- ❸ Type `mb-objdump -h executable.elf` at the prompt in the **Xygwin** shell window to list various sections of the program, along with the starting address and size of each section



6. From the objdump output, complete the following memory map table:

Note: There are many sections are listed in the output, only some of them are being asked below

Section	Starting Address	Ending Address	Size in Hex
.text			
.rodata			
.sdata2			
.data			
.sdata			
.sbss			
.bss			
.bss_stack			



7. Looking at the objdump output, list the sections that consume no memory.

---



---



Open the **TestAppLinkScr** file and change the stack size to **0x100**. Recompile the code, re-execute the objdump command, and analyze the output.

- ❶ In XPS, double-click the **TestAppLinkScr** file under **Compiler Options** in the **Applications** tab
- ❷ Change the stack size to **0x100**
- ❸ Save the linker script
- ❹ Recompile the program

- 5 Execute the **mb-objdump** command in the **Xygwin** shell



8. From the object dump output, complete the following table:

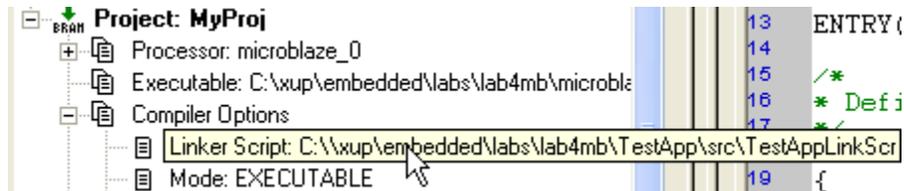
Section	Starting Address	Ending Address	Size in Hex
.bss			
.bss_stack			



Define a heap space of **256** bytes, keeping the stack size to **256** in the linker script. Add the heap space in the linker script after the stack allocation, as shown below. Recompile the code, re-execute the objdump command, and analyze the output.

```
_HEAP_SIZE = 256; /* heap size definition */
```

- 1 Change the linker script file to include a heap definition of 256 bytes near the top of the linker script as shown below. To do this, double click on the **Linker Script** under **Project: MyProj** as follows:



Add the `_HEAP_SIZE` definition near the top as shown below:

```
00
01 STACK_SIZE = DEFINED( STACK_SIZE ) ? STACK_SIZE : 0x100;
02 HEAP_SIZE = 256; /* heap size definition */
03
04 /* Define all the memory regions in the system */
05 MEMORY
```

- 2 Add the three lines shown below to the `.bss_stack` section of the linker script as shown below

```
83 .bss_stack : {
84     . = ALIGN(8);
85     _heap = .;
86     heap_start = heap;
87     += HEAP_SIZE;
88     = ALIGN(16);
89     heap_end = .;
90     . += _STACK_SIZE;
91     . = ALIGN(8);
92     _stack = .;
93     __stack = _stack;
94 } > ilmb_cntlr
95 }
96
```

- 3 Save the linker script

- ④ Recompile the code
- ⑤ Execute the **mb-objdump** command in the **Xygwin** shell

 9. From the object dump output, complete the following table:

Section	Starting Address	Ending Address	Size in Hex
.bss			
.bss_stack			

 Edit the **system.c** file to define a local variable, **k**, initialized with a value of **0**, and increment it by **1** inside the **for loop**. Recompile the code, re-execute the **objdump** command, and analyze the output.

- ① Change the **system.c** file to define a local variable and increment it in the **for loop**

The code should look like the following:

```
main() {

    XGpio gp_out;
    int j=0;
    int i=0;
    int k=0; /* add this statement */

    XGpio_Initialize(&gp_out, XPAR_LEDS_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&gp_out, 1, 0x00);

    while(1) {
        j = (j + 1) % 256;
        k = k + 1; /* add this statement */
    }
}
```

- ③ Save the **system.c** file
- ③ Recompile the code
- ④ Execute the **mb-objdump** command in the **Xygwin** shell

 10. From the object dump output, complete the following table:

Section	Starting Address	Ending Address	Size in Hex
.text			

 11. Has the **.text** section changed? Why or why not?

---



Edit the **system.c** file to add a statement that will output the value of the variable **k** to the LEDs. Recompile the code, re-execute the `objdump` command, and analyze the output.

- ❶ Change the **system.c** file to add a statement that outputs the value of variable **k** to the LEDs

The code should look like the following:

```
main() {

    XGpio gp_out;
    int j=0;
    int i=0;
    int k=0;

    XGpio_Initialize(&gp_out, XPAR_LEDS_4BIT_DEVICE_ID);
    XGpio_SetDataDirection(&gp_out, 1, 0x00);

    while(1) {
        j = (j + 1) % 16;
        k = k + 1;

        // write the value of j to the LEDs
        XGpio_DiscreteWrite(&gp_out, 1, j);
        XGpio_DiscreteWrite(&gp_out, 1, k);

        // software delay loop for pause
        for(i=0; i<10; i++);

    }

}
```

- ❷ Save the **system.c** file
- ❸ Recompile the code
- ❹ Execute the **mb-objdump** command in the **Xygin** shell



10. From the object dump output, complete the following table:

Section	Starting Address	Ending Address	Size in Hex
.text			



11. Has the `.text` section changed? Why or why not?

---

## Verifying in Hardware (Optional)

## Step 5

---



In the **system.c** file, comment out all lines which refer to or use the variable **k** and save the file. Connect the hardware board for which you have developed this lab (Spartan3 Starter Kit board). Create a bitstream file.

- ❶ Connect the included programming cable to the PC parallel port and the Spartan3 Board
- ❷ Connect a serial cable between the PC and the DB-9 connector on the board.
- ❸ Attach the included power supply to the board.
- ❹ In the source code file, comment out all lines which refer to variable **k**
- ❺ Change **i<10** to **i<400000** in the **for loop** and save the file
- ❻ Recompile the source file
- ❼ Click **Tools** → **Update Bitstream** and implement and generate the BIT file



Download the generated bitstream file into the hardware board by using a Parallel programming cable. Verify that the board is programmed and the LEDs are turning ON and OFF in proper sequence.

- ❶ Download the generated bit file by clicking **Tools** → **Download**
- ❷ After the board is programmed, you will see that the LEDs are turning ON and OFF in the desired sequence
  - **Note:** The LEDs are connected in such a way that “1” will turn OFF the LED
- ❸ Turn off the power when done

---

## Conclusion

---

XPS can be used to define, develop, and integrate the software components of the embedded system. A device driver interface can be defined for each of the peripherals and the processor. XPS creates an MSS file that represents the software side of the processor system. The peripheral-specific functional software can be developed and compiled. The executable file can be generated from the compiled object codes and libraries. The linker script can be edited to control placement of various code segments into target memory.