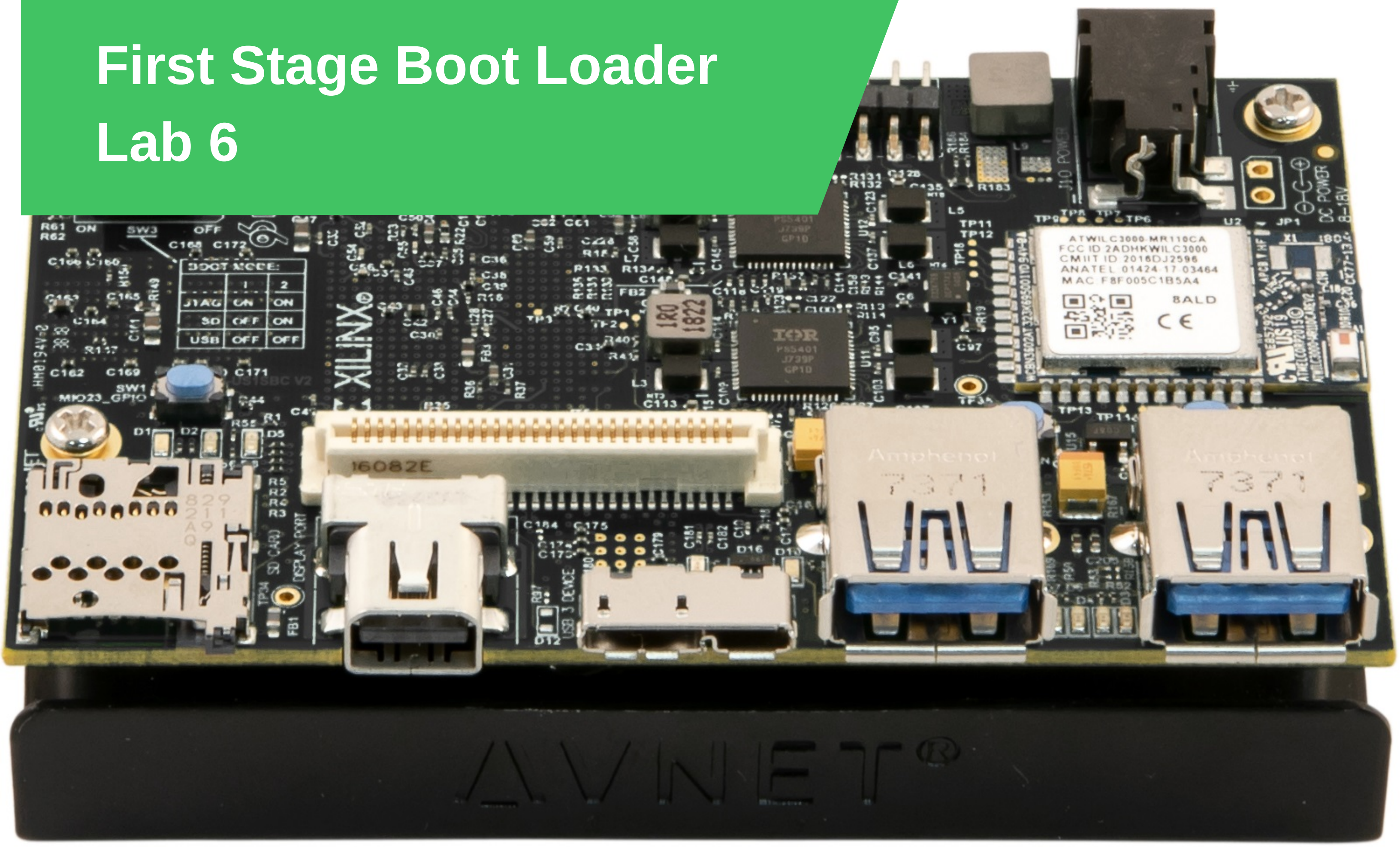
Avnet Technical Training Course

First Stage Boot Loader
Lab 6



Tools:	2019.1
Training Version:	v11
Date:	July 2019

Lab 6 Overview

In Lab 5, we used the SDK JTAG connection and a TCL script to initialize the ARM processor registers. This was convenient and worked well for debugging. However, a real product is not going to be tethered to a JTAG connection. We need a method to initialize the ARM in an embedded fashion. This is done with code called the First Stage Boot Loader or FSBL.

Xilinx SDK includes a template to automatically create the FSBL for us. By interpreting the provided hardware platform, SDK will generate a fully functional FSBL for us. This is one of the many useful things that SDK automates to make the job as a software developer a bit easier.

Lab 6 Objectives

When you have completed Lab 6, you will know:

- How to generate the FSBL
- What the FSBL application includes
- How to recognize some of the initialization sequences

Experiment 1: Generate the FSBL

Similar to the flow for creating the Test applications, SDK is used to generate a First Stage Bootloader application using a template.

Experiment 1 General Instruction:

Create a new software application. Apply the FSBL template. Note the application size and target memory. Change to the Release configuration and note size.

Experiment 1 Step-by-Step Instructions:

1. In SDK, select **File → New → Application Project**.

2. Name it **zynq_fsbl_0** and create a *new* BSP. We create a new BSP because the fsbl requires a file system library (xilffs). Of course, this could be added in the existing BSP but would be unnecessary for our memory test and peripheral test applications. The Xilffs library is automatically included in the BSP when the application type Zynq FSBL is selected. Click **Next >**.

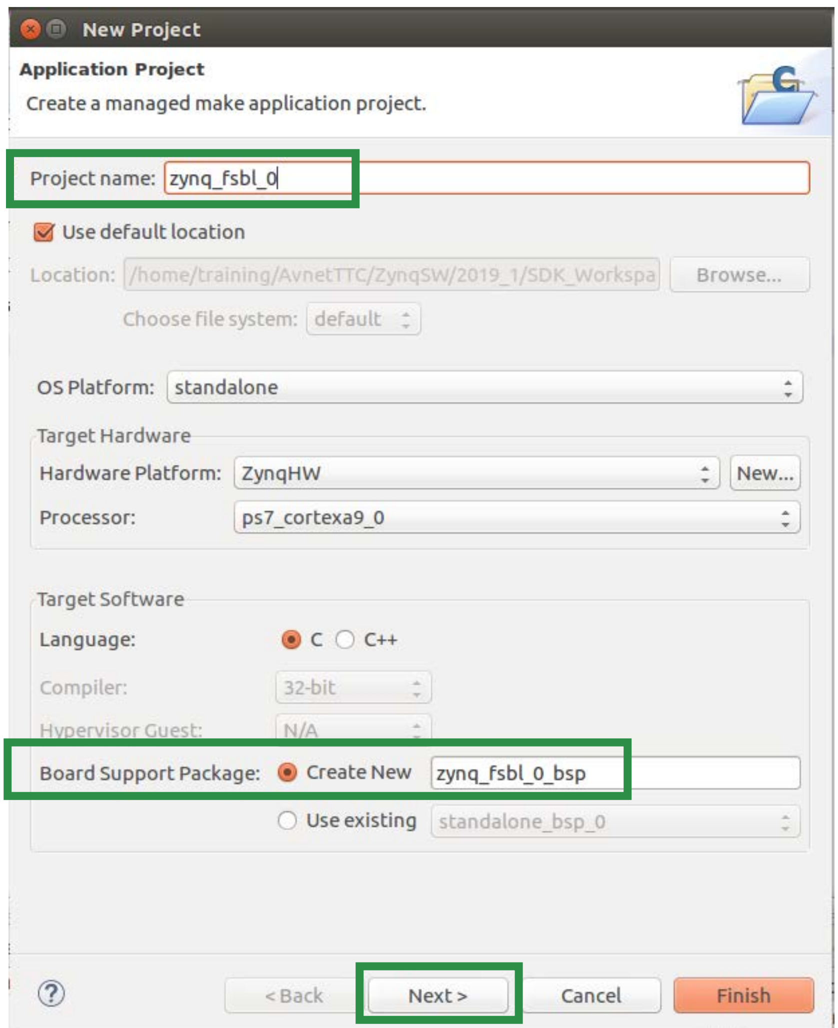


Figure 1 – FSBL Application

3. Select **Zynq FSBL** then click **Finish**.

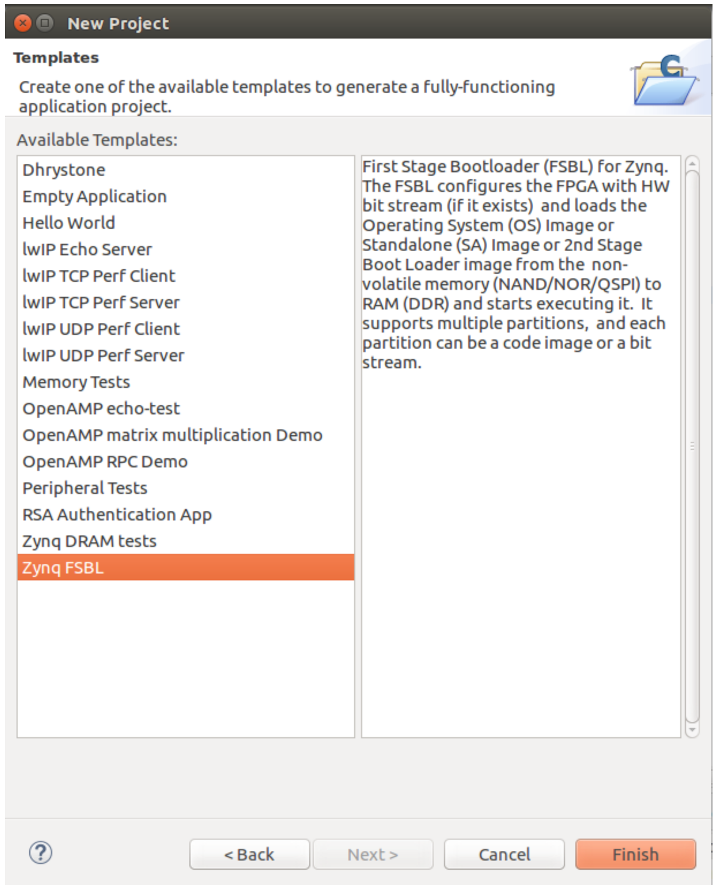


Figure 2 – Use the Zynq FSBL Template

4. Notice this message at the top of the console (emphasis added):

```
**** Build of configuration Debug for project zynq_fsbl_0 ****
```

5. Use the console output (or file zynq_fsbl_0.elf.size in the Debug directory) to determine the Debug configuration build size. Use the linker script (lscript.ld in the src directory) to determine the target memory.

Questions:

Answer the following questions:

- *What is the size of the FSBL application with the Debug configuration?*

____

- *What is the target memory for the FSBL?*

____

6. If you recall in lab 03 the BSP was modified to have stdin and stdout have a value of ps7_uart_1. We must do the same of the newly generated zynq_fsbl_0_bsp. Open the **system.mss** file from the Project explorer window, then **select Modify this BSP's Settings**.

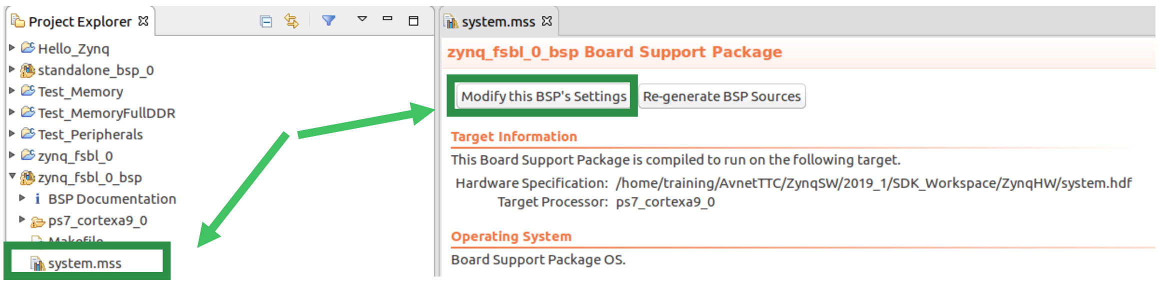


Figure 3 -- Modify BSP

7. Similarly to lab 03 **select standalone** from the Board Support Package Settings, then **change the stdin and stdout Value to ps7_uart_1.** Finally select OK to save settings and regenerate BSP.

8. You can see what the compiler settings for the Debug configuration are by right-clicking on the **Debug** folder underneath the **zynq_fsbl_0** application and selecting **Properties**.

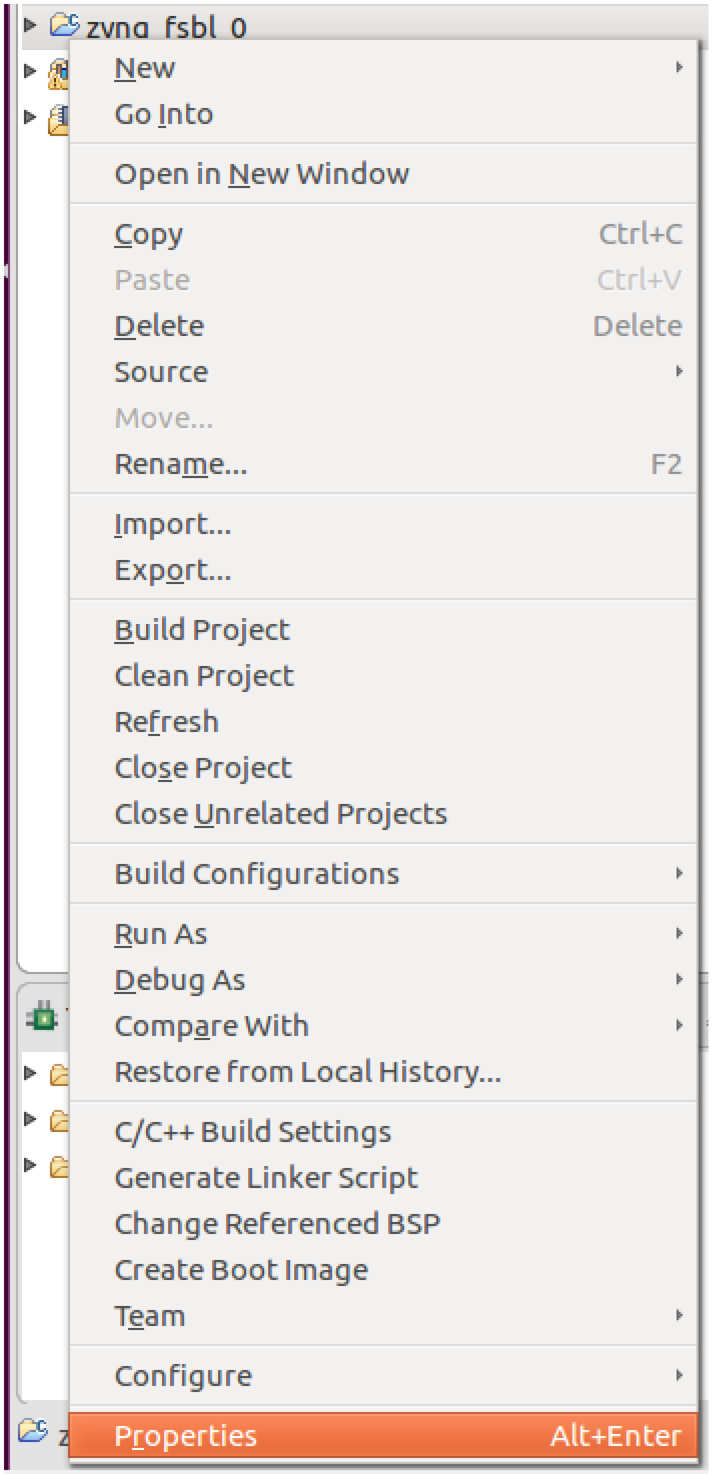


Figure 4 – See Properties for Debug Configuration

9. Expand *C/C++ Build* and select **Settings**. Then under *Tool Settings*, expand *ARM v7 gcc compiler* and look at **Optimization** and **Debugging**. Notice that the Optimization Level is at None and Debug Level is at Maximum.

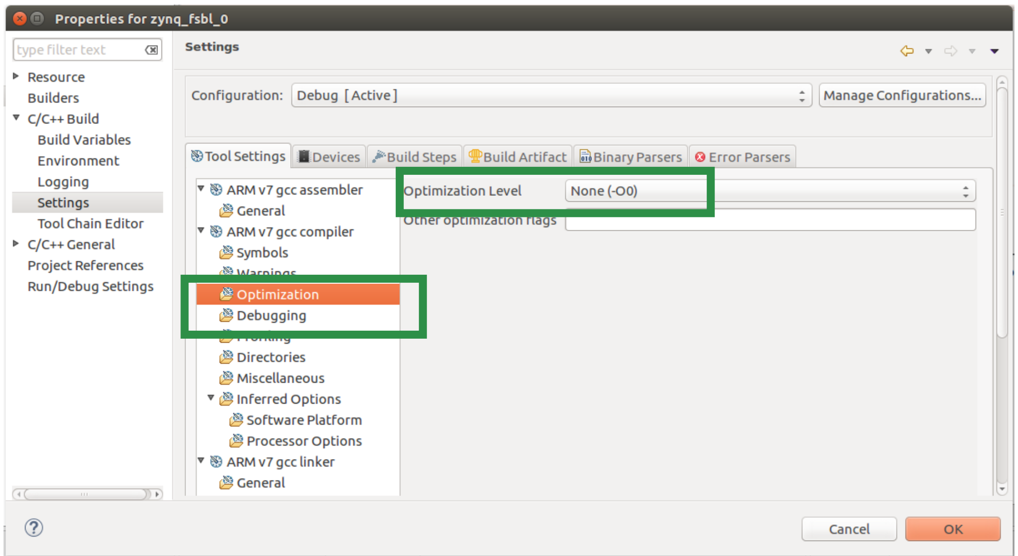


Figure 5 – Debug Configuration Optimization and Debugging

If we were going to debug this application, then these settings are perfect. However, the whole point of creating the FSBL is to move to an embedded delivery. Once we are no longer debugging, it is to our advantage to optimize the code compilation and remove the debug symbols.

10. Click the **Manage Configurations** button. You'll find this button in the upper right corner of the *Properties for Debug* dialog

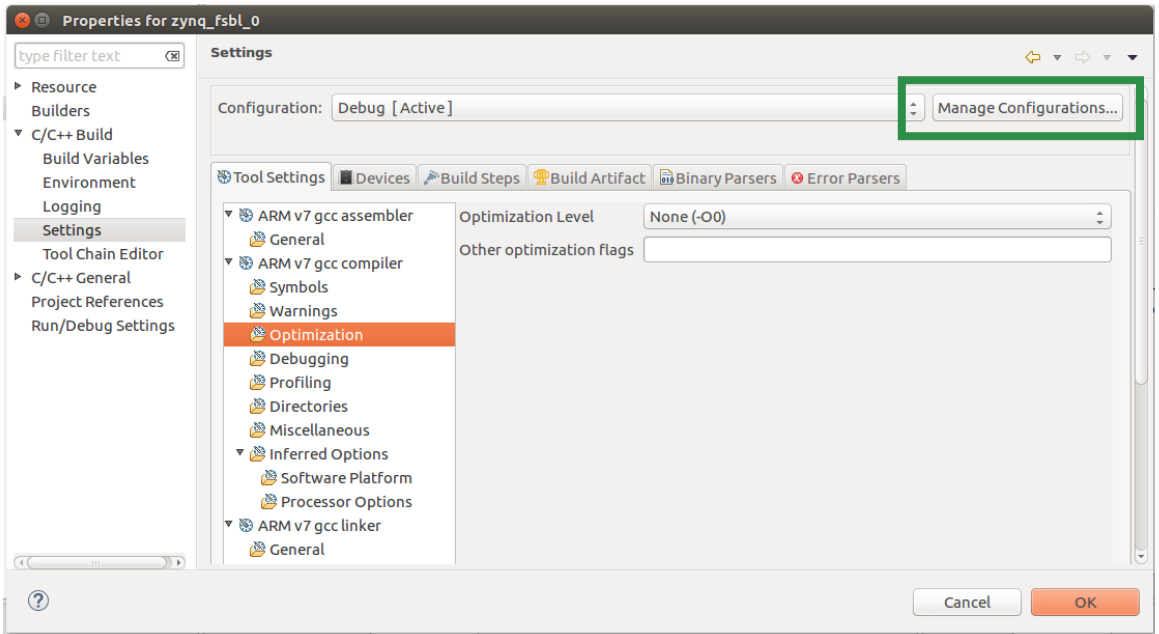


Figure 6 – Manage Configurations Button

11. Select **Release** and then click the **Set Active** button. Click **OK**.

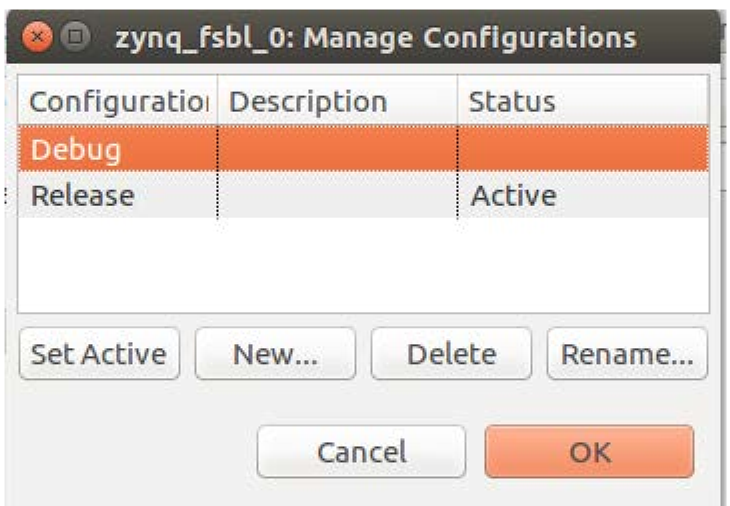


Figure 7 – Set Active Configuration to Release

12. Change the pull-down for *Configuration* to **Release [Active]**. Check the Optimization and Debugging settings for this configuration as a comparison. Click **Apply** then **OK**.

13. If it doesn't build on its own, click **Project → Build All**.

14. Notice that a **Release** sub-folder is now present in the zynq_fsbl_0 application project.

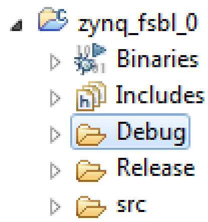


Figure 8 – Release Configuration Added and Built

Question:

Answer the following question:

- *What is the size of the FSBL application with the Release configuration?*

____

Experiment 2: Investigate the FSBL

Now that we have the FSBL generated and built, it is worth taking a minute to examine what we have!

Experiment 2 General Instruction:

Examine the FSBL sources. Open ps7_init.c and review the code.

Experiment 2 Step-by-Step Instructions:

1. Expand **zynq_fsbl_0 → src**. Notice the various .c and .h files that SDK pulled together for the FSBL.

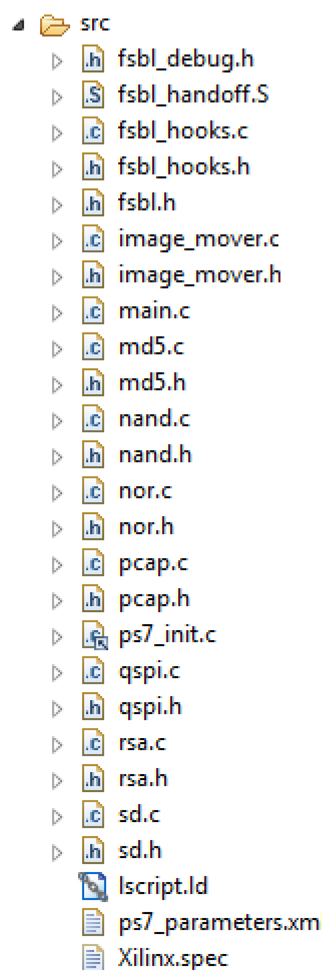


Figure 9 – FSBL Sources

2. Figuring out where function main is could be a challenge. We'll take advantage of the search function to find it. Select the **src** folder, then from the SDK pull-down menu select **Search → C/C++**. Type in `'main'` into the *Search string*. Select the radio button for *Selected resources*. Click **Search**.

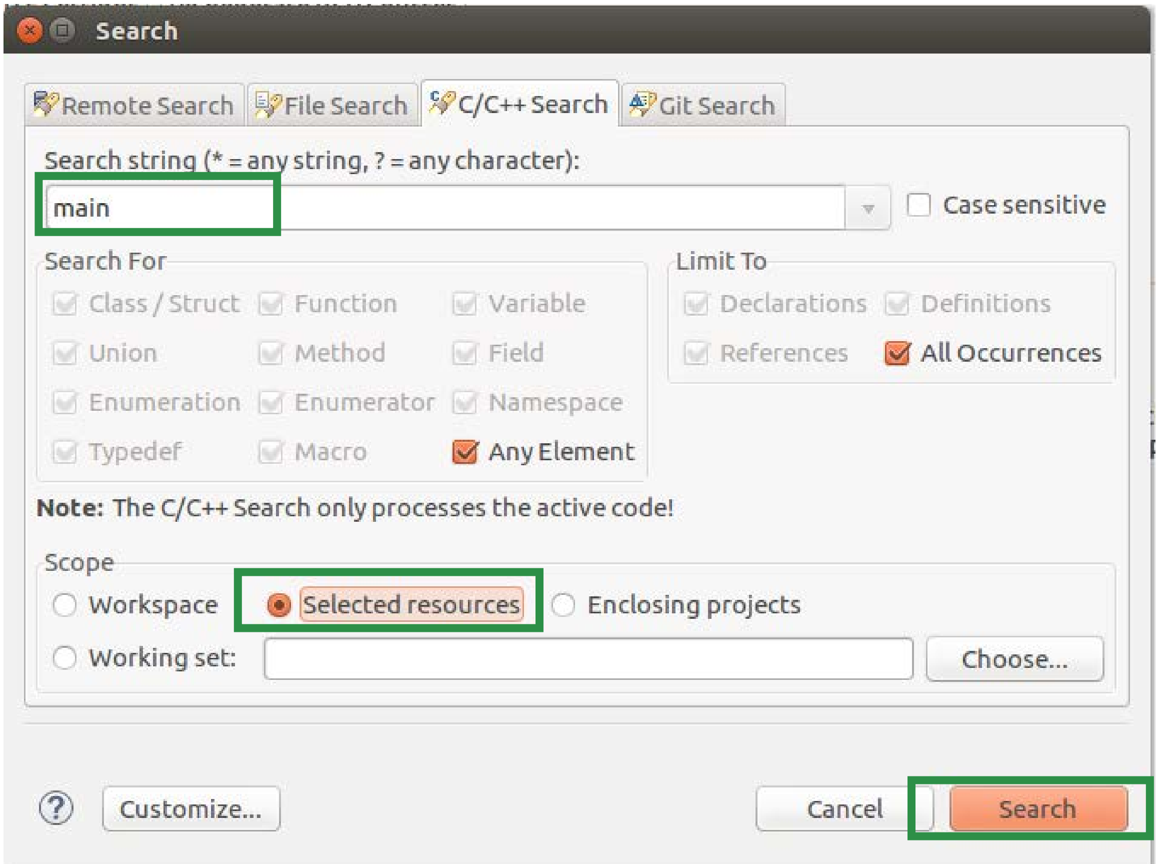


Figure 10 – Search for main

3. A new *Search* tab opens with the results. Notice that main(void) is in the file main.c. In the Search tab, double-click the result to open main.c to the main function.

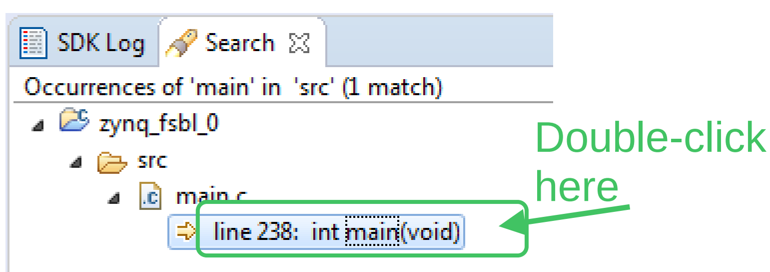


Figure 11 – Search Results

4. Scroll down through the code to get an overview of what the code does. Make use of the hover and right-click features in the editor to determine definitions and also to jump to other function declarations.

   Some Examples:

   a. Line 245 executes ps7_init(). Select it, then right-click and select **Open Declaration**. Click **No**, as we won't change the scalability settings now. It is important to note that the ps7_init.c file that was just opened is <u>large</u>.

   b. This places us near the bottom of a large file called ps7_int.c file. You can see the comments here to see what is going on:

      - Get the silicon version and make a few custom adjustments
      - Initialize the MIO
      - Initialize the PLLs
      - Initialize the clocks
      - Initialize the DDR
      - Configure the peripherals

   c. If you scroll back up to the top of the file, you will see very detailed, highly commented instances of register settings which is what all these initialize functions are performing.

   d. Close the ps7_init.c file.

   e. Back in main.c, assuming ps7_init() was successful, the next thing is to Unlock the System Level Control Registers (SLCR).

   f. Scroll down to around line 296. Since XPAR_PS7_DDR_0_S_AXI_BASEADDR is defined, the DDR Initialization Check will be run.

```
297
298 #ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
299
300     /*
301      * DDR Read/write test
302      */
303     Status = DDRInitCheck();
304     if (Status == XST_FAILURE) {
305         fsbl_printf(DEBUG_GENERAL,"DDR_INIT_FAIL \r\n");
306         /* Error Handling here */
```

   g. The PCAP (Programming Configuration Access Port) then gets set up starting on line 317.

   h. Scroll down to Line 382. Here the boot mode is read and compared to a mask. This will read the MODE jumper settings on your board.

i. Based on that BootMode setting, the FSBL then jumps to the appropriate Boot Media, which might be QSPI, NAND, NOR, SD or JTAG.

j. Once the boot media is accessed, a bitstream will be programmed to the PL if it is found, and the application ELF will be copied into the appropriate memory.

k. Lastly, a handoff address is determined, and then the FSBL hands off control to the application loaded from the Boot Media on Line 572.

Questions:

Answer the following questions:

- *In what file was the FSBL main() function?*

____

- *Which file included all of the ARM register settings? Where did it originate (prior to FSBL generation)?*

____

____

Exploring Further

If you have more time and would like to investigate more…

- Go back to Line 382 of main.c. What is the address of the BOOT_MODE_REG? Find this address in the TRM. Which register is this?

This concludes Lab 6.

Revision History

Date	Version	Revision
12 Nov 13	01	Initial release
23 Nov 13	02	Revisions after pilot
01 May 14	03	ZedBoard.org Training Course Release
30 Oct 14	04	Revised to Vivado 2014.3
31 Dec 14	05	Revised to Vivado 2014.4
09 Mar 15	06	Finalize SDK 2014.4
Oct 15	07	Updated to SDK 2015.2
Aug 16	08	Updated to SDK 2016.2
Jun 17	09	Updated to 2017.1 for MiniZed + Rebranding
Feb 18	10	Updated to Vivado/SDK 2017.4
July 19	11	Updated to Vivado/SDK 2019.1

Resources

www.minized.org

www.microzed.org

www.picozed.org

www.zedboard.org

www.xilinx.com/zynq

www.xilinx.com/sdk

www.xilinx.com/vivado

www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pdf

www.xilinx.com/support/documentation/sw_manuals/ug1046-ultrafast-design-methodology-guide.pdf

Answers

Experiment 1

- *What is the size of the FSBL application with the Debug configuration?*

159596 bytes

- *What is the target memory for the FSBL?*

ps7_ram_0_S_AXI_BASEADDR, which is the 192 KB on-chip RAM

- *What is the size of the FSBL application with the Release configuration?*

153788 bytes => 5808 bytes (or 3.9%) smaller

Experiment 2

- *In what file was the FSBL main() function?*

main.c

- *Which file included all of the ARM register settings? Where did it originate (prior to FSBL generation)?*

ps7_init.c

The imported hardware platform. Remember Lab 1?