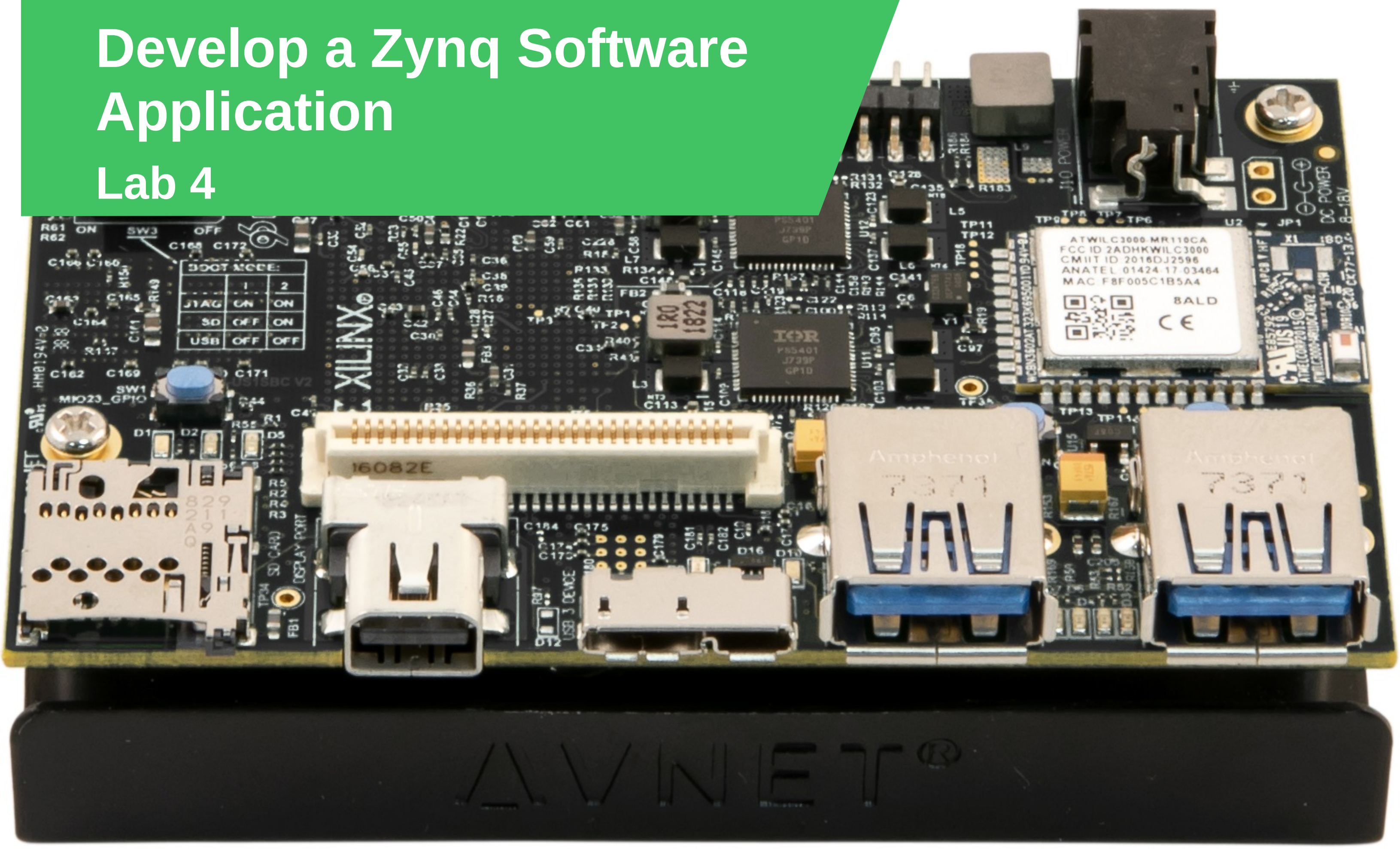
AVNET
Reach Further™

Avnet Technical Training Course

Develop a Zynq Software Application

Lab 4



Tools:	2019.1
Training Version:	v11
Date:	July 2019

Lab 4 Overview

With a Hardware Platform and BSP, we are now ready to add an application. With the BSP generated and built, drivers for all the peripherals are now ready and available in the workspace. In fact, Xilinx provides example code for most of the peripherals! We are ready to start writing our own code!

In addition to creating a brand new application, we will also take advantage of two auto-generated, example applications that will allow us to see much more sophisticated software applications than we would have time to develop during this course.

Lab 4 Objectives

When you have completed Lab 4, you will know how to:

- Add new software applications to SDK
- Use example code to target the UART in a Hello World application
- Apply example project templates, including
    - Memory Tests
    - Peripheral Tests
- Identify application code size and location
- Modify linker scripts to change the target memory location

Experiment 1: Develop an Application with Example Code

SDK is now ready for software development. Getting started can be difficult. You may have inherited a hardware platform that you are still working to understand. You may not be familiar with the drivers that Xilinx has provided in the generated BSP. Reading through all the driver code to get started could be overwhelming. It would be extremely helpful to start with example code and then make modifications.

That is exactly what we will do in this experiment. Hopefully this will help you get in the habit of looking for free example code provided by Xilinx any time you encounter a new peripheral and driver.

Experiment 1 General Instruction:

Add an Empty software application. Copy and paste the Hello World example code provided with the UART driver. Generate the linker script to target the application memory region to the PS7 on-chip RAM0. Determine the code size and location.

Experiment 1 Step-by-Step Instructions:

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

2. In the **Project Name** field type in `Hello_Zynq`. Change the **BSP** to the existing Standalone BSP. Click **Next >**.

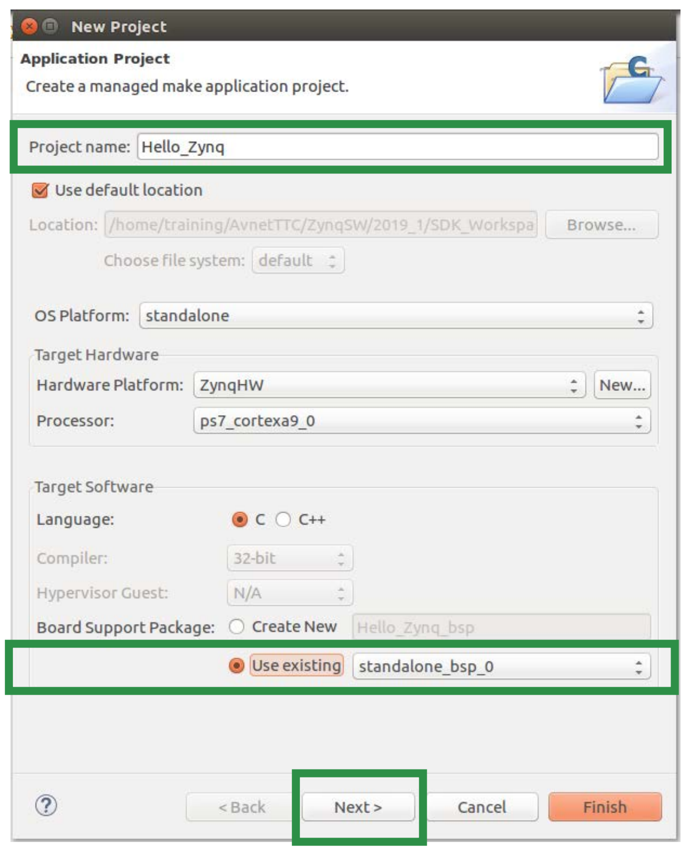


Figure 1 - New Application Wizard

3. You can see in the *Available Templates* that there is a Hello World template. We are NOT going to use this right now, although it would work perfectly fine. Instead, select **Empty Application**. Click **Finish**.

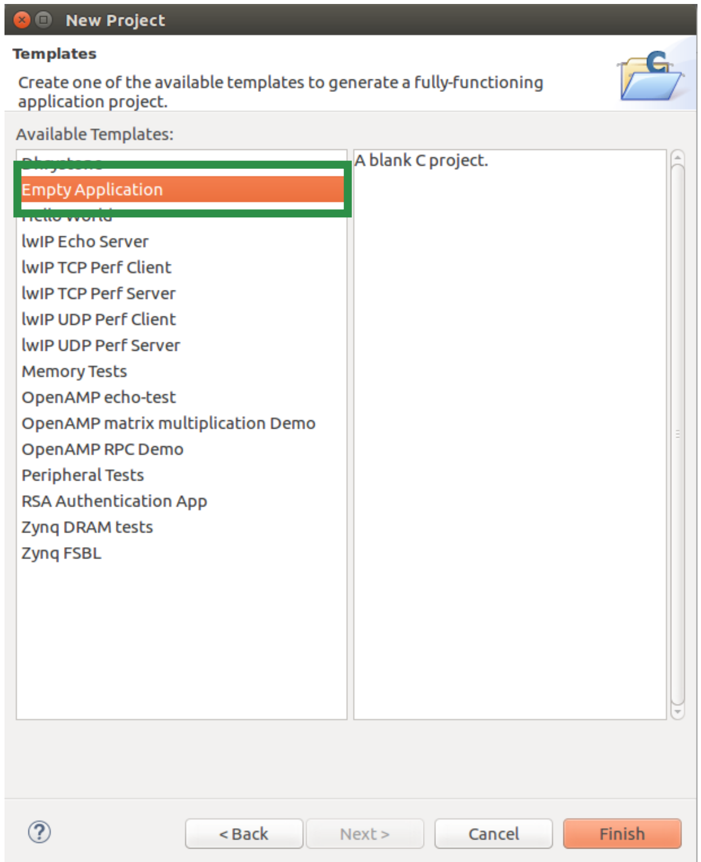


Figure 2 – Empty Application

4. Notice that the Hello_Zynq application is now visible in *Project Explorer*. Notice the different icons that represent C Software Application , BSP , and Hardware Platform .

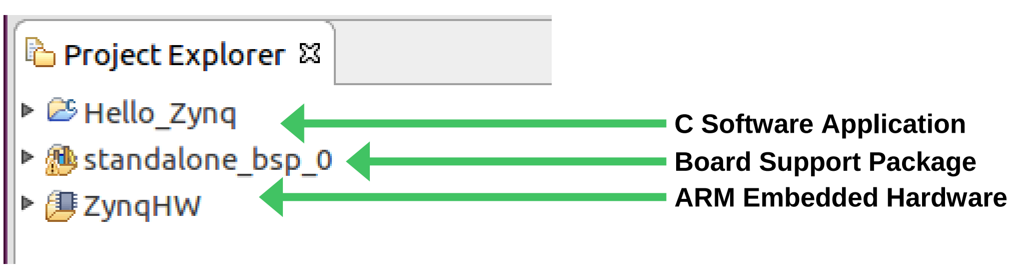


Figure 3 – Project Explorer View with Hello World C Application Added

5. In *Project Explorer*, browse to **Hello_Zynq → src**. You will see that you have been given two files of note. The file `lscript.ld` is a default linker script. The `README.txt` states

Empty application. Add your own sources.

6. As instructed, we will add our own source now. Right-click on the **src** folder, then select **New → Source File**. In the dialog, give the source file the name **hello_zynq.c**, then click **Finish**.

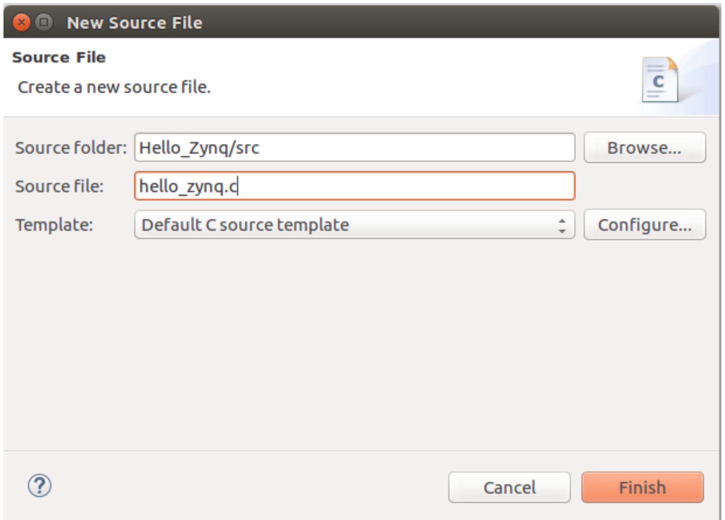


Figure 4 – New Source File

7. By default, SDK will build the application automatically after it is added. This can be changed by deselecting **Project → Build Automatically**, but we will not do that now. The results of the build are available in the Console window at the bottom of the SDK GUI. In this case, the build fails because the new hello_zynq.c has nothing in it other than a commented header.

Figure 5 – Auto Build of Hello_Zynq Fails

8. Another artifact of the automatic build is that the BSP now has an error because it couldn't find a main() in the new application. You will see the BSP icon with a red X on it now. We'll fix this in a minute after we have a main().

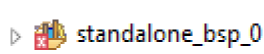


Figure 6 – BSP Error Since the New Source has no main()

9. Next, we'll locate some example code. Using a file explorer navigate to

   /home/training/SDK/2019.1/data/embeddedsw/XilinxProcessorIPLib/drivers/uartps_v3_7/ examples.

10. This opens the directory where the example source files are located. **xuartps_hello_world_example.c** is provided, along with several other examples that would likely prove useful in future development including polled, interrupt, and echo examples. Right Click on **xuartps_hello_world_example.c**. and open the file using **Open with → gedit**

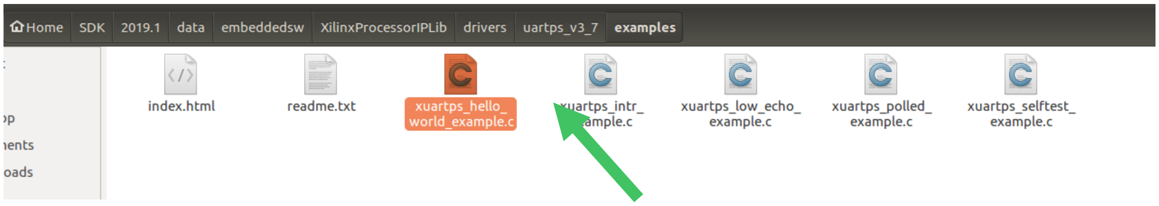


Figure 7 – Example UART C Files Driver Directory

11. Use Ctrl-A or click-drag to select the contents of the entire file. Then use Ctrl-C to copy. Switch back to SDK, close the import examples window and use Ctrl-V to paste into hello_zynq.c. Take a moment to browse through the code to understand what it is doing.

12. Use **File → Save**, or the Save icon, or Ctrl-S to save the file. The application automatically builds.

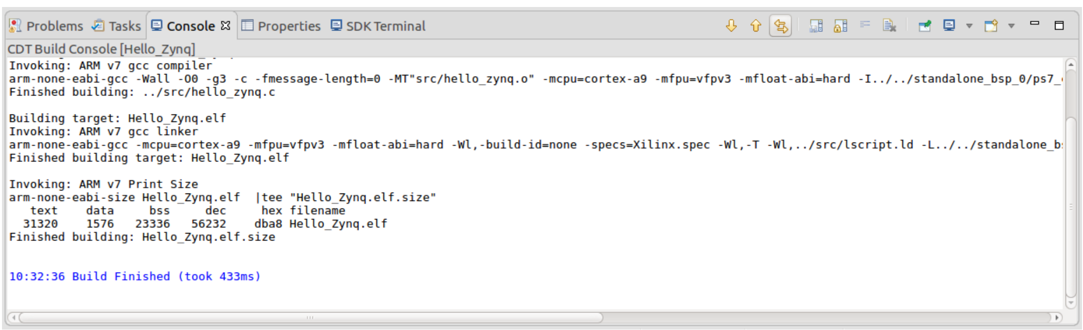


Figure 8 – Hello World Application Automatically Built

13. Notice at the bottom of the report that the application size is reported. This is printed from the report file **`Hello_Zynq.elf.size`** which can also be found in *Project Explorer* under **Hello_Zynq → Debug → Hello_Zynq.elf.size**. Note that "Debug" refers to the Build Configuration, which by default is always Debug for a new application. Note that this application is 56,232 bytes large.

    Be aware that you may see small deviations to these sizes reported as it is dependent on the exact compiler version and options you are using.

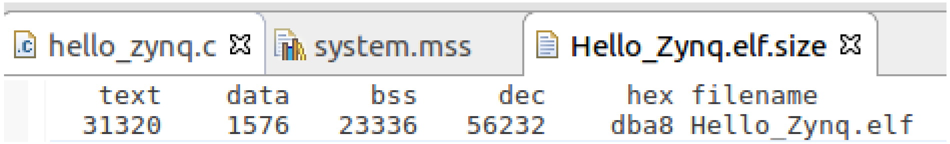


Figure 9 – Size of the Hello_Zynq Application

14. Now that we have a main() in the new application, we can correct the issue with the BSP. SDK should be able to fix itself, but in case it doesn't, do the following. Select **standalone_bsp_0** in *Project Explorer*. Now select **Project → Clean**. Select the radio button for *Clean projects selected below*. Check the box for **standalone_bsp_0**. Click **OK**. After cleaning, an automatic rebuild will start which will eliminate the red X from the BSP.

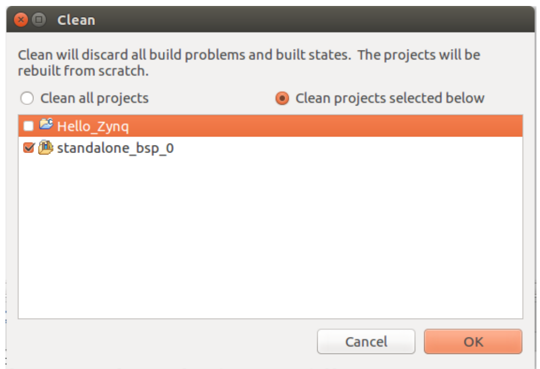


Figure 10 – Clean and Rebuild the BSP

15. The lscript.ld is the linker script for this application. This is useful as a report to see what memory was targeted for the application. It may also be edited to change the location of the application. Double-click **Hello_Zynq → src → lscript.ld** now. Notice that the window opens to a graphical view that is indicated by the *Summary* tab in the lower left corner. We'll look at the *Source* tab in a minute.

16. The first *Summary* section shows the *Available Memory Regions*.

    a. **axi_bram_ctrl_0_Mem0** is blockRAM located in the PL

    b. **ps7_ddr_0** is the DDR3L

    c. **ps7_qspi_linear_0** is the QSPI

    d. **ps7_ram_0** is 192 KB of on-chip RAM

    e. **ps7_ram_1** is 64 KB of on-chip RAM

Available Memory Regions

Name	Base Address	Size
axi_bram_ctrl_0_Mem0	0x40000000	0x2000
ps7_ddr_0	0x100000	0x1FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFF0000	0xFE00

Figure 11 – Available Memory Regions for Application

17. The second *Summary* section shows the *Stack and Heap Sizes*. The default size is 8KB each for Stack and Heap.

Stack and Heap Sizes

Stack Size 0x2000

Heap Size 0x2000

Figure 12 – Stack and Heap Sizes

18. The third *Summary* section shows the mapping between the application Sections and the Memory Region. You'll see in this case that all 29 different sections were all assigned to DDR3L.

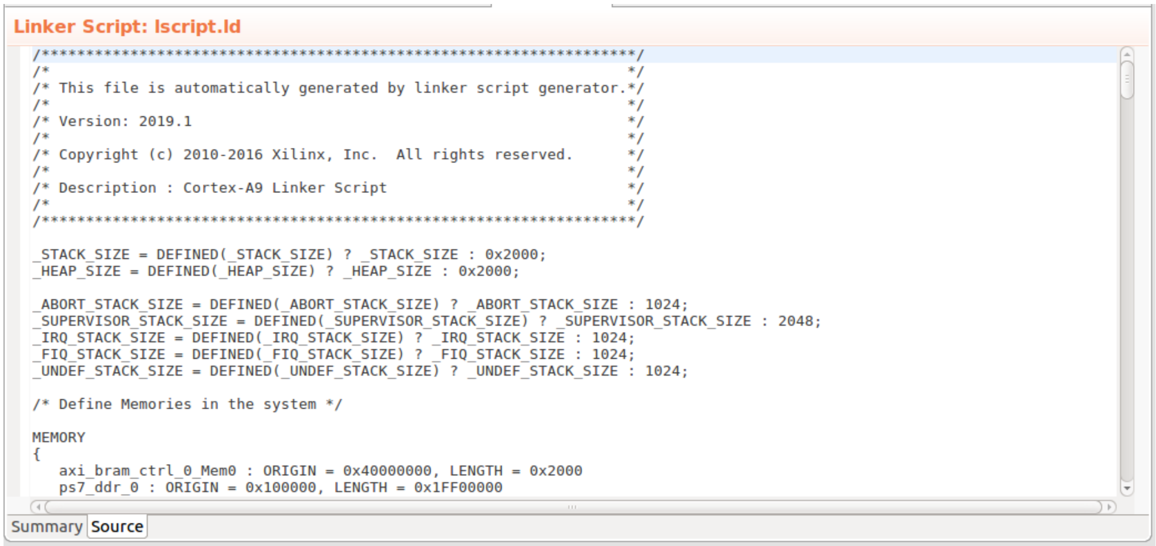
Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0
.data	ps7_ddr_0
.data1	ps7_ddr_0
.got	ps7_ddr_0
.ctors	ps7_ddr_0
.dtors	ps7_ddr_0
.fixup	ps7_ddr_0
.eh_frame	ps7_ddr_0
.eh_framehdr	ps7_ddr_0
.gcc_except_table	ps7_ddr_0
.mmu_tbl	ps7_ddr_0
.ARM.exidx	ps7_ddr_0
.preinit_array	ps7_ddr_0
.init_array	ps7_ddr_0
.fini_array	ps7_ddr_0
.ARM.attributes	ps7_ddr_0
.sdata	ps7_ddr_0
.sbss	ps7_ddr_0
.tdata	ps7_ddr_0
.tbss	ps7_ddr_0
.bss	ps7_ddr_0
.heap	ps7_ddr_0
.stack	ps7_ddr_0

Figure 13 – Section to Memory Region Mapping

This *Summary* is editable. You could easily change the Stack and Heap sizes here. You could also change the *Memory Region* for one section at a time. If you wanted to change the target Memory Region for all sections, this would be very tedious here.

19. Switch to the *Source* tab.


Linker Script: lscript.ld

```
/*******************************************************************/
/*                                                                 */
/* This file is automatically generated by linker script generator.*/
/*                                                                 */
/* Version: 2019.1                                                 */
/*                                                                 */
/* Copyright (c) 2010-2016 Xilinx, Inc.  All rights reserved.      */
/*                                                                 */
/* Description : Cortex-A9 Linker Script                           */
/*                                                                 */
/*******************************************************************/

_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x2000;
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x2000;

_ABORT_STACK_SIZE = DEFINED(_ABORT_STACK_SIZE) ? _ABORT_STACK_SIZE : 1024;
_SUPERVISOR_STACK_SIZE = DEFINED(_SUPERVISOR_STACK_SIZE) ? _SUPERVISOR_STACK_SIZE : 2048;
_IRQ_STACK_SIZE = DEFINED(_IRQ_STACK_SIZE) ? _IRQ_STACK_SIZE : 1024;
_FIQ_STACK_SIZE = DEFINED(_FIQ_STACK_SIZE) ? _FIQ_STACK_SIZE : 1024;
_UNDEF_STACK_SIZE = DEFINED(_UNDEF_STACK_SIZE) ? _UNDEF_STACK_SIZE : 1024;

/* Define Memories in the system */

MEMORY
{
   axi_bram_ctrl_0_Mem0 : ORIGIN = 0x40000000, LENGTH = 0x2000
   ps7_ddr_0 : ORIGIN = 0x100000, LENGTH = 0x1FF00000
```

Summary | Source

Figure 14 – Source for Linker Script

Now you see the source linker script ***code*** from which the Summary was generated. This view is also editable. If you wanted to change all sections to the on-chip RAM, you could do a global search and replace here. That is straight-forward, and many will be comfortable editing the linker script in this manner.

20. Close lscript.ld.

21. Another method to modify the linker script is to generate a completely new one using a wizard. This is possible with a tool provided in the SDK. Right-click on **Hello_Zynq** in the *Project Explorer* and select **Generate Linker Script**.

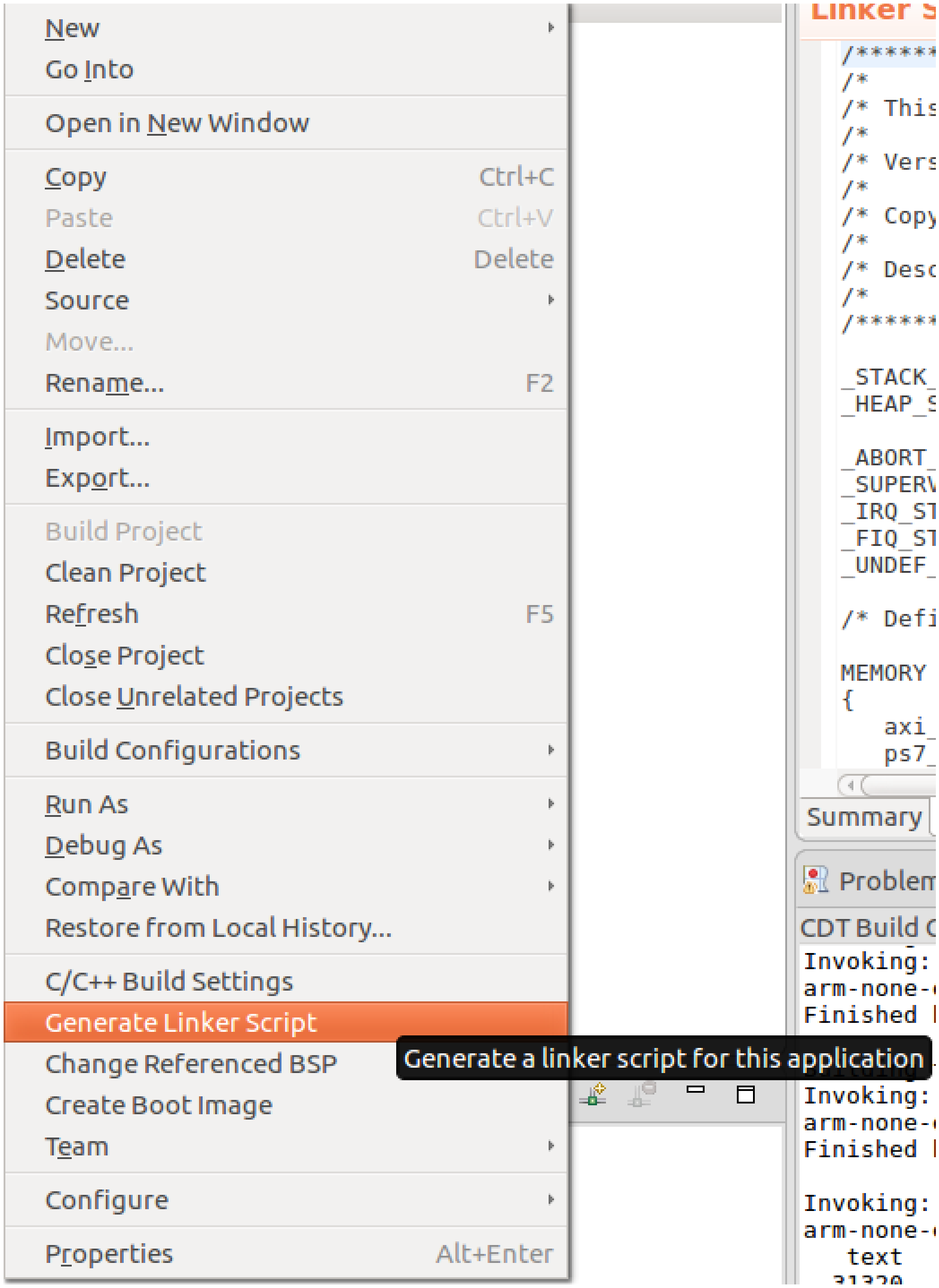
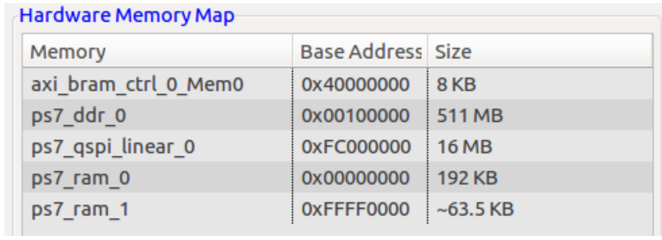


Figure 15 – Generate Linker Script

22. Notice that in the *Hardware Memory Map*, you see the same five memories that were in the lscript.ld *Summary*.



Hardware Memory Map

Memory	Base Address	Size
axi_bram_ctrl_0_Mem0	0x40000000	8 KB
ps7_ddr_0	0x00100000	511 MB
ps7_qspi_linear_0	0xFC000000	16 MB
ps7_ram_0	0x00000000	192 KB
ps7_ram_1	0xFFFF0000	~63.5 KB

Figure 16 – Hardware Memory Map

23. On the right side in the *Basic* tab, set all of the sections to ***ps7_ram_0***. This Linker Script Generator defaults to 1 KB for *Heap Size* and *Stack Size* which is smaller than the 8KB created when Hello_Zynq was generated, but 1KB is acceptable. If you wanted to change this, type in the number of bytes (for example, 2048 rather than 2 KB). The *Advanced* tab allows you to be much more specific about assigning particular pieces within the Code and Data section, but we won't do that today. Click **Generate** and then **Yes** to allow overwriting the existing linker script.

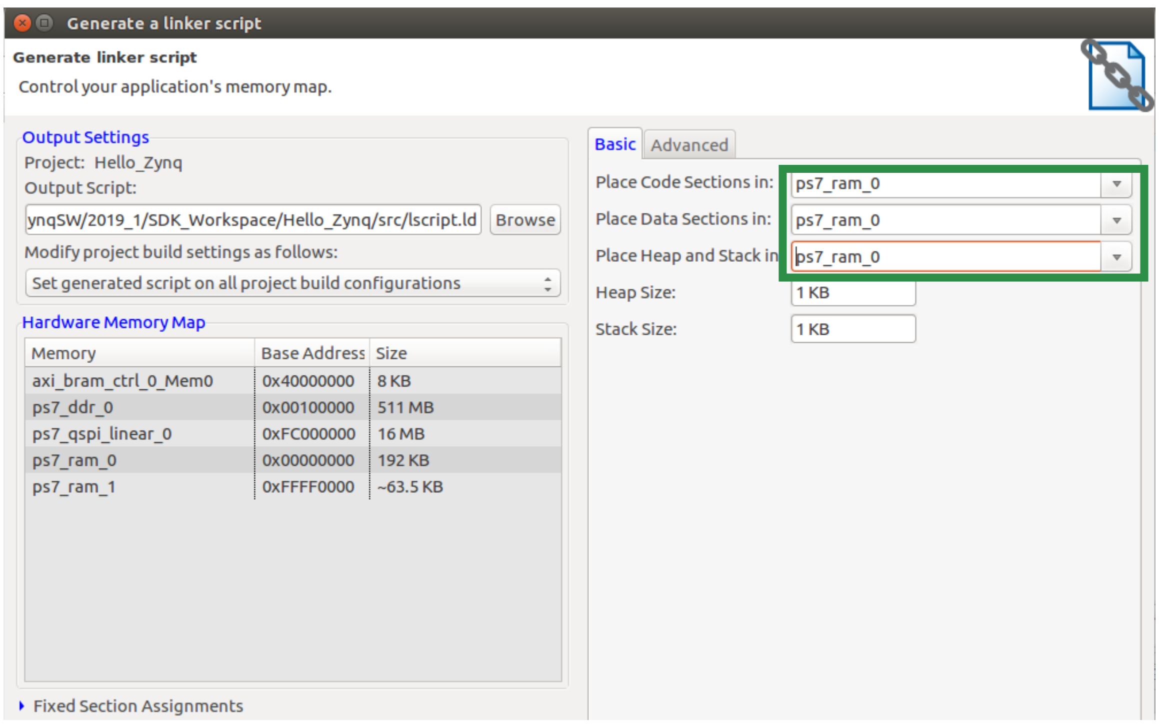


Figure 17 – Generate Linker Script for Memory Tests

24. Notice that SDK will automatically rebuild the Hello_Zynq application based on the new linker script. The new size is ~41,896bytes. It makes sense since both the stack and heap were each reduced by 7 KB each.

25. Open the newly generated lscript.ld to see the changes implemented by the Generate Linker Script tool.

26. Close lscript.ld and any other files related to Hello_Zynq.

Question:

Answer the following question:
• *You've been assigned a task to develop code to test reading and writing to the PL BRAM (peripheral axi_bram_ctrl_0 in this hardware platform). What do you do?*
____
____
____

Experiment 2: Add Peripheral Test

It would be nice to test the various peripherals in the hardware platform. SDK provides a template for testing the peripherals. This peripheral test exercises all of the peripherals and their associated controllers built inside the SoC.

Experiment 2 General Instruction:

Add the Peripheral Test application. Determine the size and target location.

Experiment 2 Step-by-Step Instructions:

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

2. In the **Project Name** field type in **Test_Peripherals**. Change the **BSP** to the existing standalone_bsp_0. By default, a new application project selects the option to generate its own BSP. If we allowed the tool to do this, the new BSP will be identical to the one already created. If at any point we wanted to change a setting in the BSP, we would have to change it in multiple BSPs, and all of them would have to rebuild. It is much better to have only one BSP, unless you are purposely making unique ones for a reason. Click **Next >**.

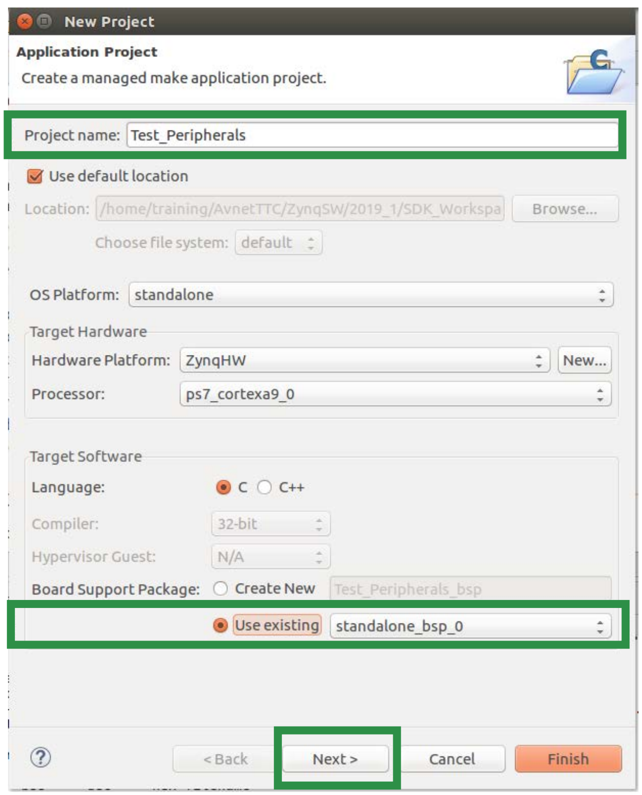


Figure 18 - New Application Wizard

3. Select **Peripheral Tests** from the *Available Templates* field. Click **Finish**.

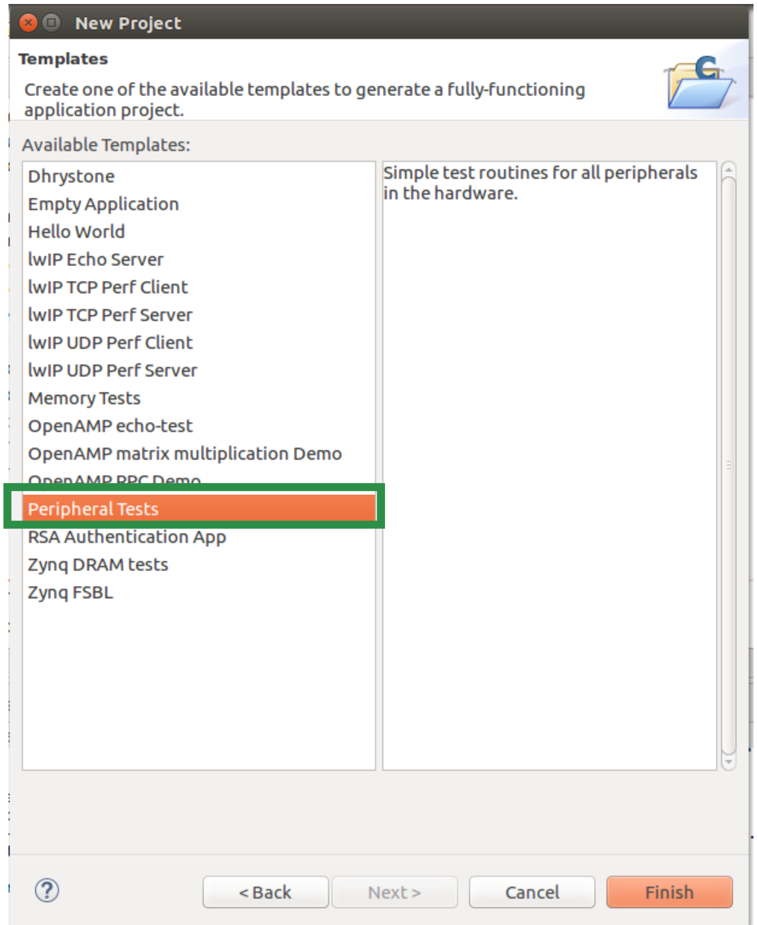


Figure 19 – New Application Project: Peripheral Tests

Questions:

Answer the following questions:

- *To what memory region(s) is the Test_Peripherals application targeted?*

____

4. The Peripheral Test application has an example of enabling the I- and D-caches. Open source file `testperiph.c`, which is the source file containing main(). Right click in the left margin of the file and select show line numbers.

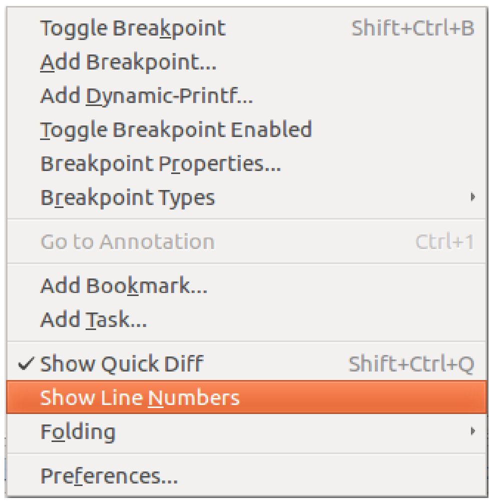


Figure 20 – Toggle Show Line Numbers

5. Notice the following code on lines 63 & 64 :

```
Xil_ICacheEnable();
Xil_DCacheEnable();
```

6. You might wonder if these enable functions are enabling the L1 cache, the L2 cache, or both. Select one of them, then right-click and select **Open Declaration**.

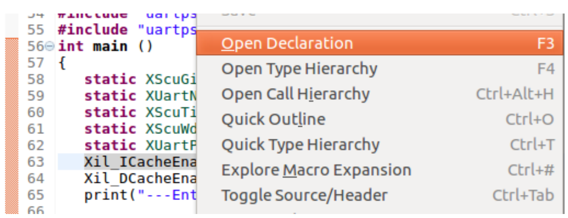


Figure 21 – Open Cache Declaration

7. The function declaration is displayed in source code file `xil_cache.c`, which is part of the BSP. You'll notice that both CacheEnable() functions include the commands to enable both the L1 and L2 caches.

```c
void Xil_ICacheEnable(void)
{
    Xil_L1ICacheEnable();
#ifndef USE_AMP
    Xil_L2CacheEnable();
#endif
}
```

Figure 22 – Declaration Shows Both L1 and L2 Enabled

Experiment 3: Add and Edit Memory Test

Another useful application template that the SDK provides is a Memory Test. This is a very useful test for any new hardware system to make sure the memory is stable prior to running an O/S.

Experiment 3 General Instruction:

Add the Memory Test application. Determine the size and target location. Copy the project. Edit the copied Memory Test to expand the test region to the entire memory.

Experiment 3 Step-by-Step Instructions:

1. Repeat steps 1 through 3 of Experiment 2 to add a new application, called **Test_Memory** with the *Memory Tests* template applied.

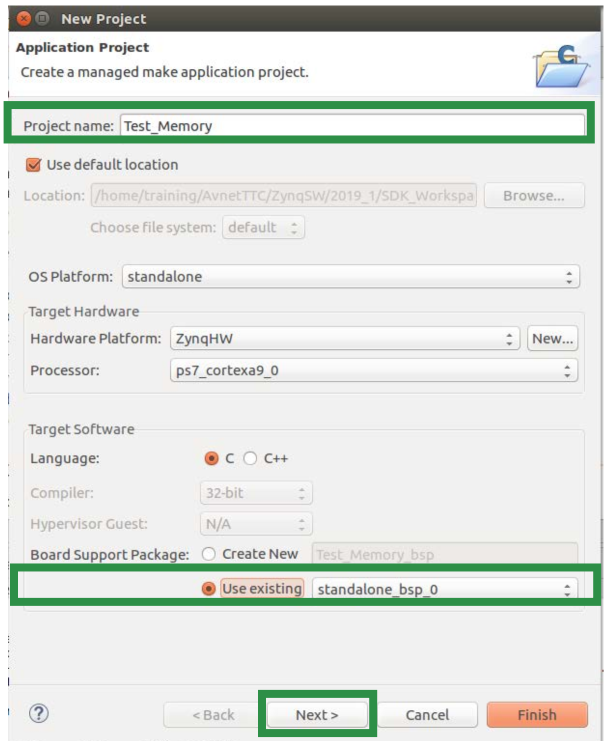


Figure 23 – Adding New Application Project Test_Memory

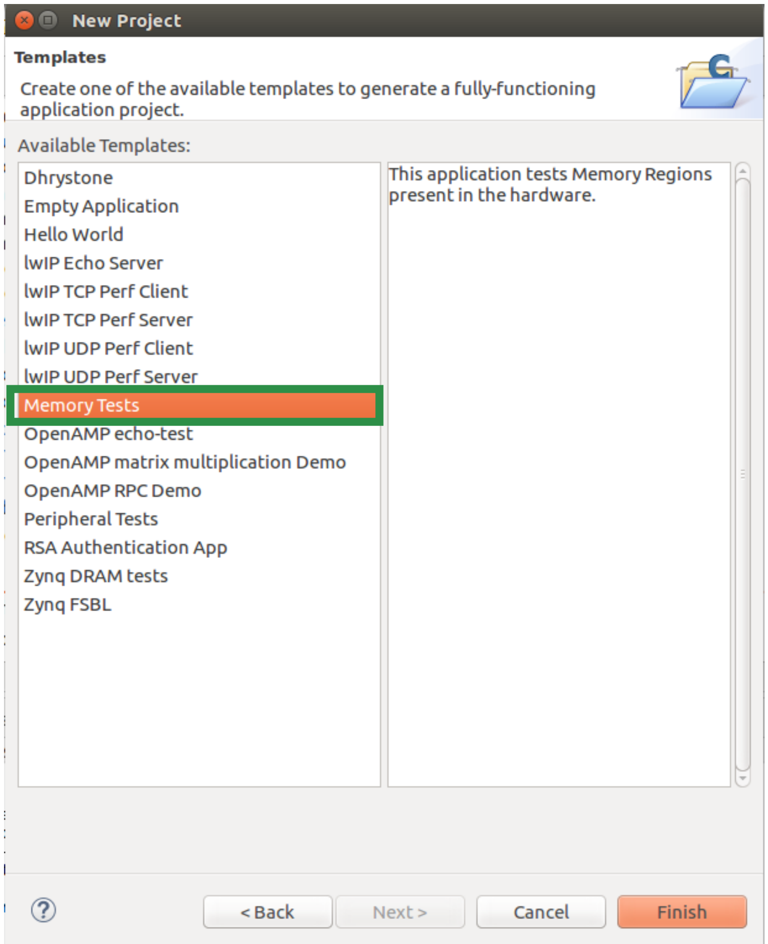


Figure 24 – Apply the Memory Tests Template

Use lscript.ld files to answer the following question.

Question:

Answer the following question:

- *To what memory region(s) is the Test_Memory application targeted?*

____

2. Open the `memorytest.c` source file. In main(), you will notice that a *for* loop exercises function test_memory_range() across n_memory_ranges memories.

3. Inside the test_memory_range() declaration, you'll see that three different tests are run – one 32-bit test, one 16-bit test, and one 8-bit test. Hover over the Xil_TestMem8() call to see the function prototype. Notice that the second parameter is the number of words to be tested.

```
85      status = Xil_TestMem8((u8*)range->base, 4096, 0xA5, XIL_TESTMEM_ALLMEMTESTS);
86      print("   extern s32 Xil_TestMem8(u8 *Addr, u32 Words, u8 Pattern, u8 Subtest); D!"); print("\n\r");
                                                                        Press 'F2' for focus
87
88  }
```

4. Inside main() Hover over n_memory_ranges and memory_ranges[] **(around line 100)** to see what values are assigned. Note: you can click into the hover pop-up window to scroll down if the contents are not completely displayed.

Questions:

Answer the following questions:

- *How much memory is tested by default?*

____

- *How many memories are tested? Which ones? Which is not? Why?*

____

5. Copy the entire Test_Memory application project by right-clicking and selecting **Copy**. Then right-click again in the whitespace underneath the *Project Explorer* and select **Paste.**

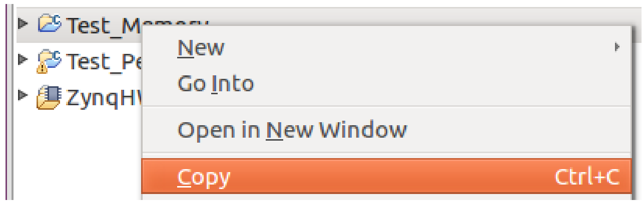


Figure 25 – Copy Test_Memory

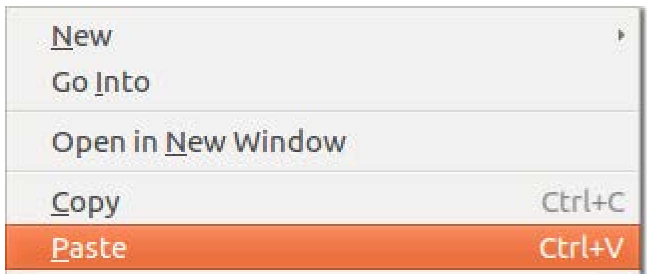


Figure 26 – Paste Test_Memory

6. Type **Test_Memory_FullDDR** as the *Project name* then click **OK**.

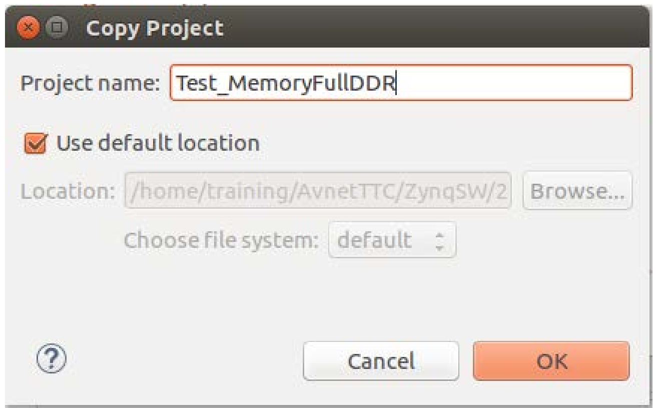


Figure 27 – Copy Project

7. Expand the *Test_Memory_FullDDR* application in the Project Explorer and then the *src* folder. Double-click on memorytest.c.

8. Scroll to where function `test_memory_range` is defined. This function tests the same 4096-byte chunk three different ways: 1024 words in 32-bit mode, 2048 half-words in 16-bit mode, and 4096 bytes in 8-bit mode. Since the memory is 512 MB, you could modify the code to test the full region. To illustrate increasing the test window (although not to the full 512 MB region in the interest of test time), change the code to test 1 MB (1,048,576), as shown below. Save the code using Ctrl-S on your keyboard. The application will automatically rebuild.

```
status = Xil_TestMem32((u32*)range->base, 1048576/4, 0xAAAA5555,
XIL_TESTMEM_ALLMEMTESTS);

status = Xil_TestMem16((u16*)range->base, 1048576/2, 0xAA55,
XIL_TESTMEM_ALLMEMTESTS);

status = Xil_TestMem8((u8*)range->base, 1048576, 0xA5,
XIL_TESTMEM_ALLMEMTESTS);
```

9. Since the on-chip RAMs are much less than 1 MB, we don't want to run a 1 MB test those RAMs. Edit `memory_config_g.c` as this :

- Move the ps7_ddr_0 section to the first range in the memory_range_s structure :

```
struct memory_range_s memory_ranges[] = {
    {
        "ps7_ddr_0",
        "ps7_ddr",
        0x00100000,
        535822336,
    },
    {
        "axi_bram_ctrl_0",
        "axi_bram_ctrl",
        0x40000000,
        8192,
```

- set **n_memory_ranges** to 1 rather than 3. Save the file.

Exploring Further

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

- Examine the remainder of the Test Peripherals source code to determine which peripherals will be tested.

This concludes Lab 4.

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 for Vivado 2014.3
31 Dec 14	05	Revised for Vivado 2014.4
12 Mar 15	06	Finalize for SDK 2014.4
Oct 15	07	Updated to Vivado 2015.2
Aug 16	08	Updated to Vivado 2016.2
Jun 17	09	Updated to Vivado 2017.1 for MiniZed + Rebranding
Jan 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

- *You've been assigned a task to develop code to test reading and writing to the PL BRAM (peripheral axi_bram_ctrl_0 in this hardware platform). What do you do?*

When starting to work with a new peripheral and its associated driver, the best place to start is the example code provided by Xilinx. Go to the system.mss Overview. Find the BRAM peripheral. Click on the Examples to get to the example code.

Experiment 2

- *To what memory region(s) is the Test_Peripherals application targeted?*

DDR3 for everything – Code, Data, heap, and stack

Experiment 3

- *To what memory region(s) is the Test_Memory application targeted?*

ps7_ram_0_S_AXI_BASEADDR for the Code, Data, heap, and stack sections

- *How much memory is tested by default?*

4096 bytes

- *How many memories are tested? Which ones? Which is not, Why?*

Three memories are tested: DDR3, axi_bram, and ram_1.

ram_0 is not tested as this is where the code, data, heap and stack reside.