

# Developing Zynq Software with Xilinx SDK

## Lab 9

### Dual Processor Software Development



November 2013  
Version 03

## Lab 9 Overview

The Zynq®-7000 All Programmable SoC contains two Cortex®-A9 processors which share common memory and peripherals. These processor cores can be configured to concurrently run independent software stacks or executable code. Asymmetric multiprocessing (AMP) is a mechanism that allows both processors to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources.

This lab describes a method of starting up both processors, each running its own bare-metal software application, and allowing each processor to communicate with the other through shared memory.

**IMPORTANT NOTE:** *This lab makes use of the ability to configure SDK for using a local repository. If at any point after this lab, you decide to use a lab solution, be sure to also configure SDK for using a local repository by following the Experiment 2, Steps 1 and 2 as well.*

## Lab 9 Objectives

When you have completed Lab 9, you will know how to do the following:

- Create a bare-metal application with its own standalone environment targeted for the first processor core CPU0
- Create a second bare-metal application with its own standalone environment targeted for the second processor core CPU1
- Generate a bootable solution which loads each application onto the respective processor core
- Debug the execution of each application from the respective processor core

## Experiment 1: Creating Bare-Metal Application Project for CPU0

This experiment shows how to create the application ELF that runs on CPU0 after the FSBL copies the application ELFs to DDR memory.

### Experiment 1 General Instruction:

Launch Xilinx Software Development Kit (SDK) and open the SDK workspace found in the following folder:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\**

Create a new empty application project and import the CPU0 application found in the **Support\_documents** folder.

### Experiment 1 Step-by-Step Instructions:

1. Launch Xilinx Software Development Kit (SDK) if not already open. **Start → All Programs → Xilinx Design Tools → Vivado 2013.3 → SDK → Xilinx SDK 2013.3.**



Figure 1 – The SDK Application Icon

2. Set or switch the workspace to the following folder and then click the **OK** button:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\**

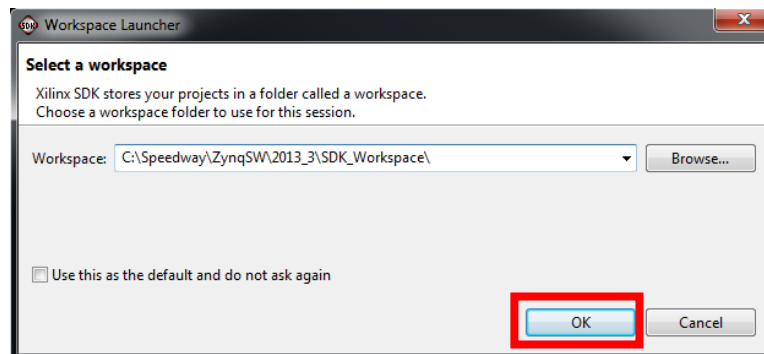
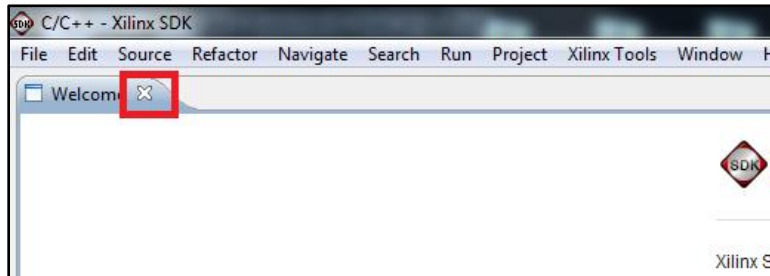


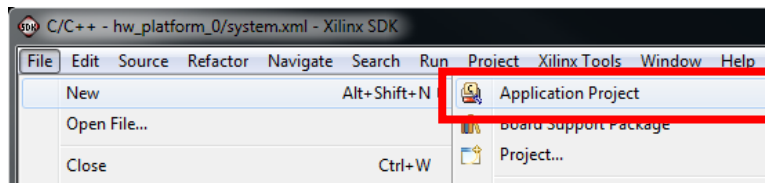
Figure 2 – Switching to the Appropriate SDK Workspace

3. Close the Welcome screen if it appears in the SDK window by clicking on the **X** control in the tab.



**Figure 3 – Closing the SDK Welcome screen**

4. Create a new SDK software application project by selecting the **File→New→Application Project** menu item.



**Figure 4 – Creating a New C Application Project**

5. In the **New Project** wizard, change the **Project name** field to **app\_cpu0** and leave the other settings to their default values. Click the **Next** button to continue.

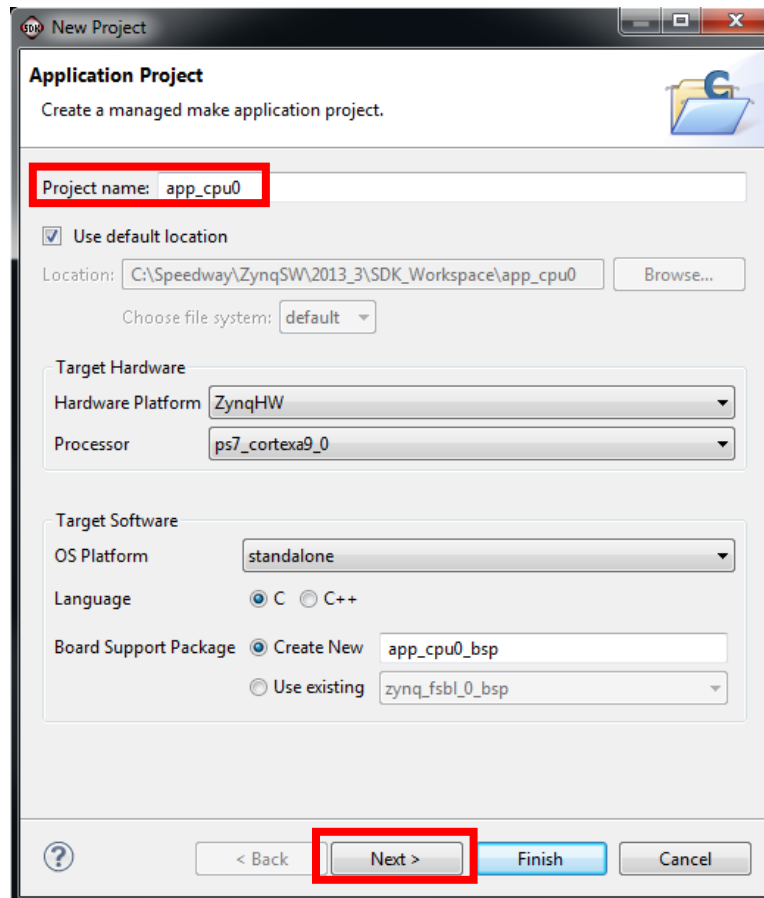
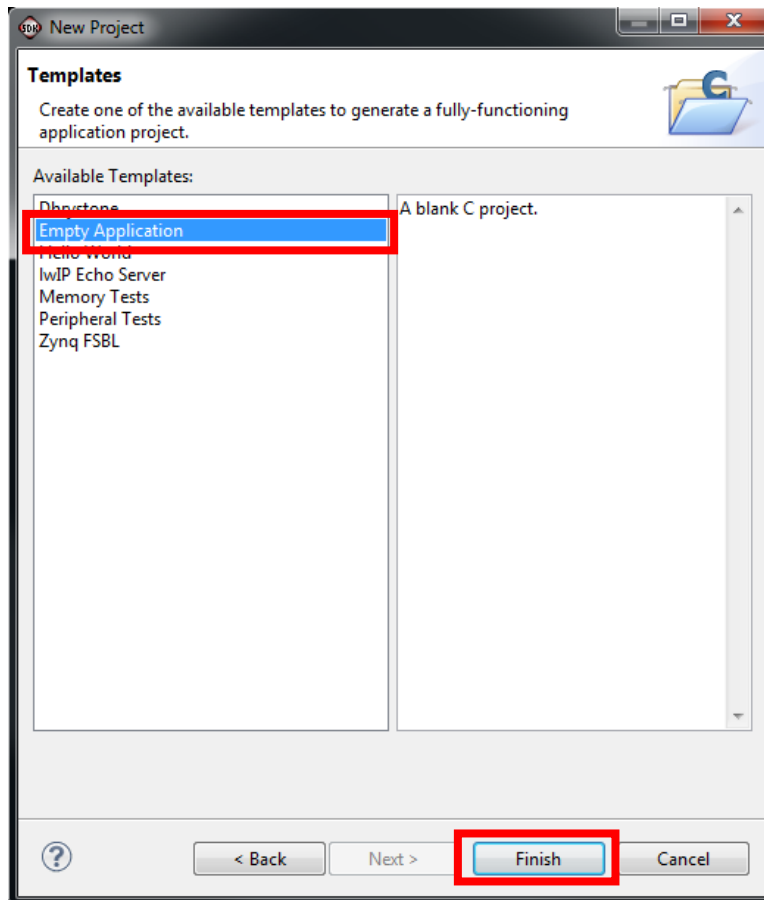


Figure 5 – Creating the CPU0 Application

6. Select the **Empty Application** project template and click the **Finish** button to complete the new project creation process using the **Empty Application** project template.

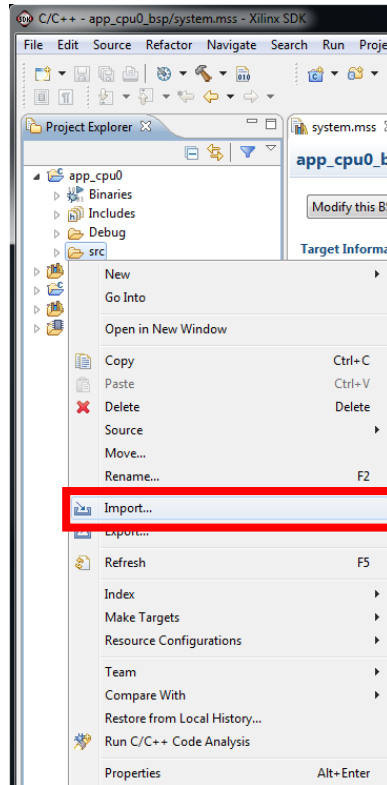


**Figure 6 – Using the Empty Application Project Template**

7. The application project is created along with the corresponding BSP **app\_cpu0\_bsp** and each of the projects will be compiled automatically.

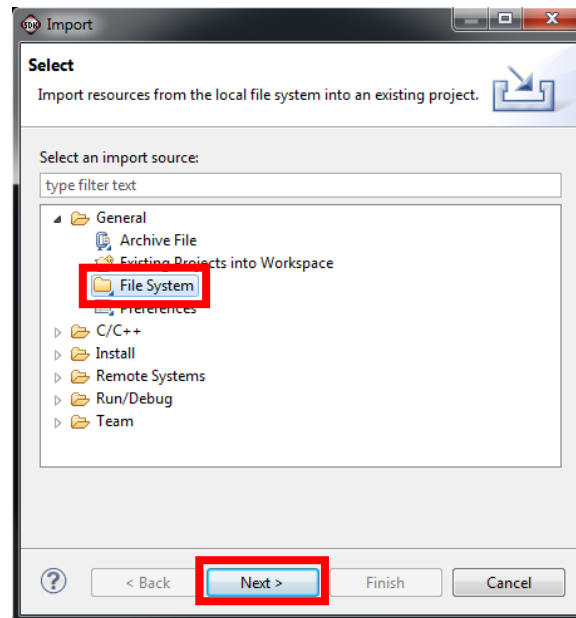
In the **Project Explorer** tab, expand **app\_cpu0** and right-click on the **src** folder.

Click on the **Import** option in the pop up menu.



**Figure 7 – Import CPU0 Application Code**

8. In the **Import** window, expand the **General** item, select the **File System** option, and click the **Next** button.



**Figure 8 – Importing from a File System**

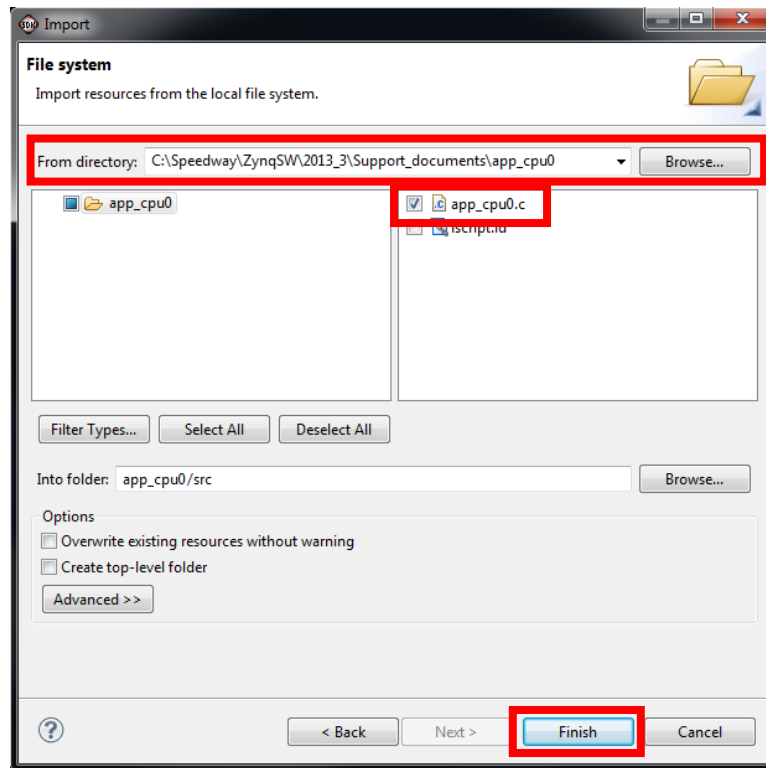


- Click on the **Browse** button and select the following folder which contains the application code that we wish to run on CPU0:

**C:\Speedway\ZynqSW\2013\_3\Support\_documents\app\_cpu0\**

After this folder is selected within the **Browse** dialog, click the **OK** button to search the folder for files to import into the application project.

Select only the **app\_cpu0.c** source file by checking the box next to the file name and then click the **Finish** button to complete the **Import** operation.



**Figure 9 – Selecting Application Source Files**

10. Open the existing **app\_cpu0** project Linker Script **lscript.ld** file. This project was generated automatically when the empty application project was created. The entire memory space from 0x00100000 to 0x1FFFFFFF is assigned to this application which is not only more than is needed for our AMP application, but we also need some separate memory space for the CPU1 application to execute out of later.

Also, if you were to look at the **app\_cpu0.c** source code you will find that the CPU1 application memory space is expected to start at 0x00200000 and there is another shared memory region which starts at address 0x03000000 so unless source code modifications are made, we will want to avoid using those memory areas when loading the CPU0 application code.

Modify the DDR memory region definition within **lscript.ld** by setting the size of the region to **0x00100000** such that **app\_cpu0.elf** occupies the memory space from 0x00100000 to 0x001FFFFFFF instead of the entire DDR memory space.

Save the changes to **lscript.ld** to rebuild the **app\_cpu0** project with the new memory space definition.

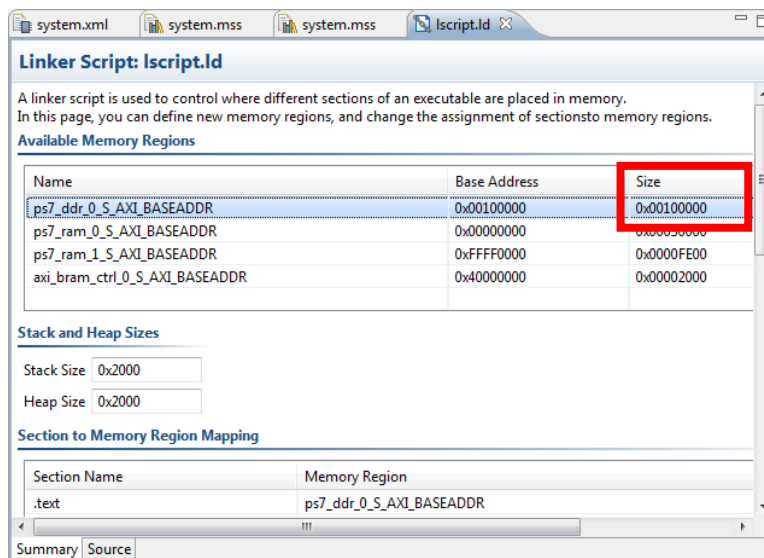
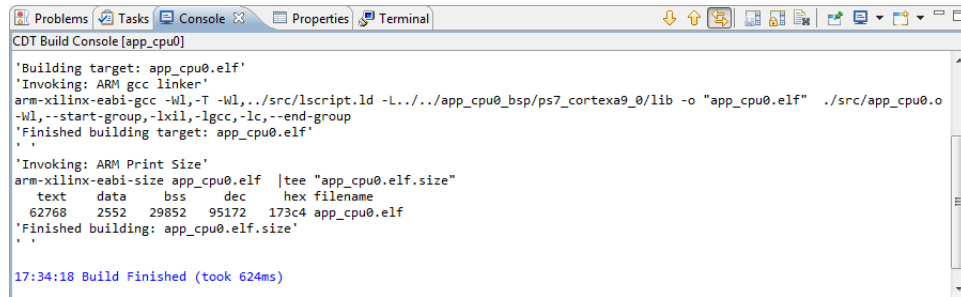


Figure 10 – Modifying lscript.ld

11. The SDK Console panel shows the results of the build. Make sure that the application is built without errors.



```
CDT Build Console [app_cpu0]

'Building target: app_cpu0.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T -Wl,../src/lscript.ld -L../app_cpu0_bsp/ps7_cortexa9_0/lib -o "app_cpu0.elf" ./src/app_cpu0.o
-Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: app_cpu0.elf'
'
'
'Invoking: ARM Print Size'
arm-xilinx-eabi-size app_cpu0.elf |tee "app_cpu0.elf.size"
  text  data   bss   dec   hex filename
 62768  2552  29852  95172  173c4 app_cpu0.elf
'Finished building: app_cpu0.elf.size'
'
'

17:34:18 Build Finished (took 624ms)
```

**Figure 11 – Application Build Console Window**

12. After SDK finishes compiling the new application code, the ELF is available in the following location:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\app\_cpu0\Debug\app\_cpu0.elf**

### **Questions:**

**Answer the following questions:**

- *What is the difference between a normal standalone application running on CPU0 and an AMP application running on CPU0?*

---

## Experiment 2: Creating Bare-Metal Application Project for CPU1

This experiment shows how to create the application ELF that runs on CPU1 after the FSBL copies the application ELFs to DDR memory.

This step is slightly different than the previous experiment in creating the application for CPU0 because the CPU1 application uses a customized BSP which is adapted from Xilinx XAPP1079 – *Simple AMP Bare-Metal System Running on Both Cortex-A9 Processors*. An important aspect of this design is that it prevents CPU1 from re-initializing shared resources. In this example application, CPU1 does not use L2 cache which is a shared resource and CPU0 “owns” this resource. If CPU1 were to use L2 cache, L2 cache flushes and invalidates would need to be requested from CPU0 and CPU0 would then exercise the action. It is beyond the scope of this Exercise to include a communication channel that enables CPU1 to request L2 cache interactions.

SDK is used to create the BSP using the customized standalone BSP from the repository which is included with the lab files.

### Experiment 2 General Instruction:

Configure SDK to use a new local repository.

Create a new Board Support Package to use an AMP Standalone BSP obtained from the local repository.

Create a new empty application project and import the CPU1 application found in the **Support\_documents** folder.

### Experiment 2 Step-by-Step Instructions:

1. Configure SDK to add the local repository which contains the AMP Standalone BSP.

Open the SDK repository preferences configuration by selecting the **Xilinx Tools→Repositories** menu item.

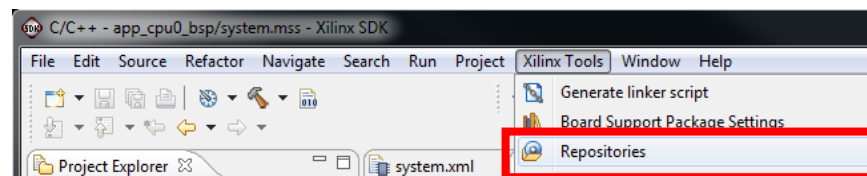


Figure 12 – Configuring SDK Repositories

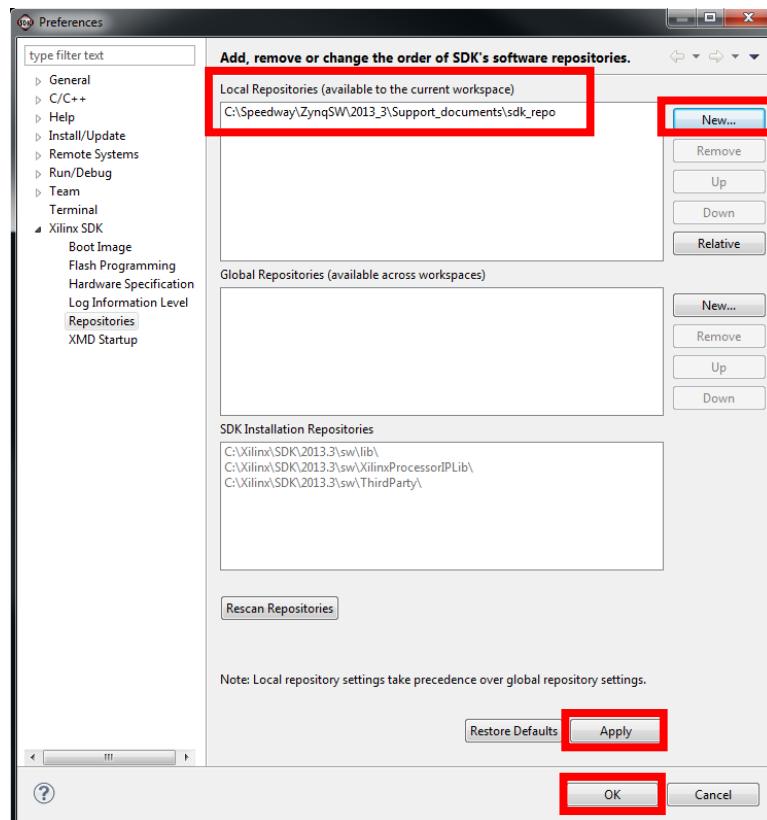
2. Add a new Local Repository entry by clicking the **New** button beside the **Local Repositories** listing.

Browse to the following folder which contains the local repository containing the AMP Standalone BSP:

**C:\Speedway\ZynqSW\2013\_3\Support\_documents\sdk\_repo\**

After this folder is selected within the **Browse For Folder** dialog, click the **OK** button to add the folder to the Local Repositories listing.

Click the **Apply** and then the **OK** buttons to close the SDK preferences window.



**Figure 13 – Adding a Local Repository**

3. Create a new SDK Board Support Package by selecting the **File→New→ Board Support Package** menu item.

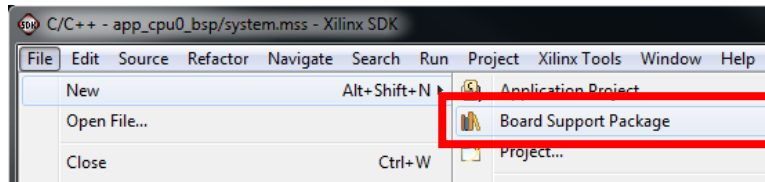


Figure 14 – Creating a New Board Support Package

4. In the **New Board Support Package Project** window, change the **Project name** field to **app\_cpu1\_bsp** to match the CPU1 application which will be created in later steps.

Use the drop down menu to change the **CPU** setting to the **ps7\_cortexa9\_1** option.

Change the **Board Support Package OS** to the **standalone\_amp** option which comes from the new local repository which was added earlier in Step 2.

Click the **Finish** button to create the new AMP Standalone BSP.

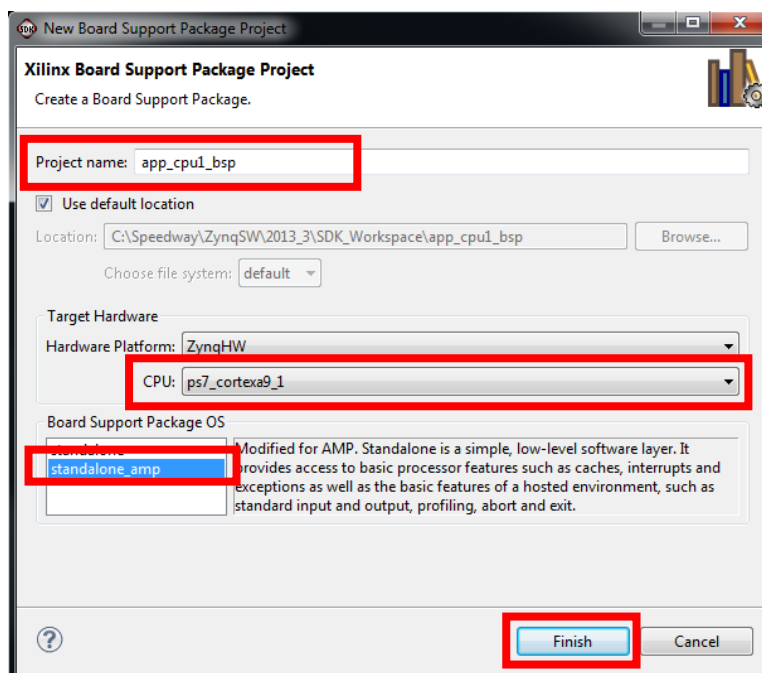


Figure 15 – New AMP Board Support Package Project

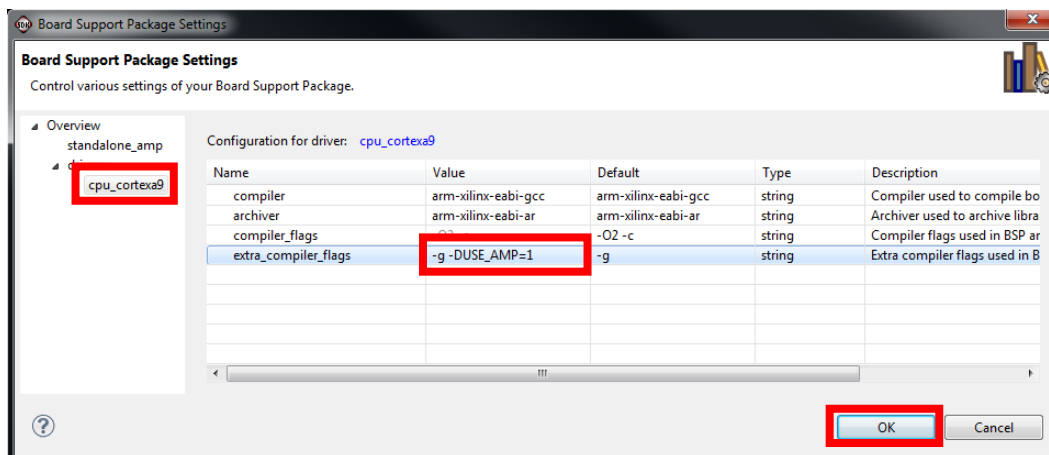
5. In the **Board Support Package Settings** window for the newly created **app\_cpu1\_bsp** project, navigate to the **cpu\_cortexa9** driver configuration under the **Overview→Drivers** item listing.

Add the following compiler flag string to the existing **extra\_compiler\_flags** property:

**-DUSE\_AMP=1**

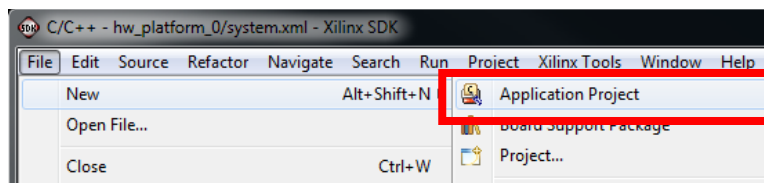
This compiler flag definition enables some code sections which prevent the re-initialization of shared CPU resources such as the SCU and L2 cache.

Click on the **OK** button to accept the new driver settings for the Board Support Package.



**Figure 16 – Creating a New C Application Project**

6. Create a new SDK software application project by selecting the **File→New→Application Project** menu item.



**Figure 17 – Creating a New C Application Project**

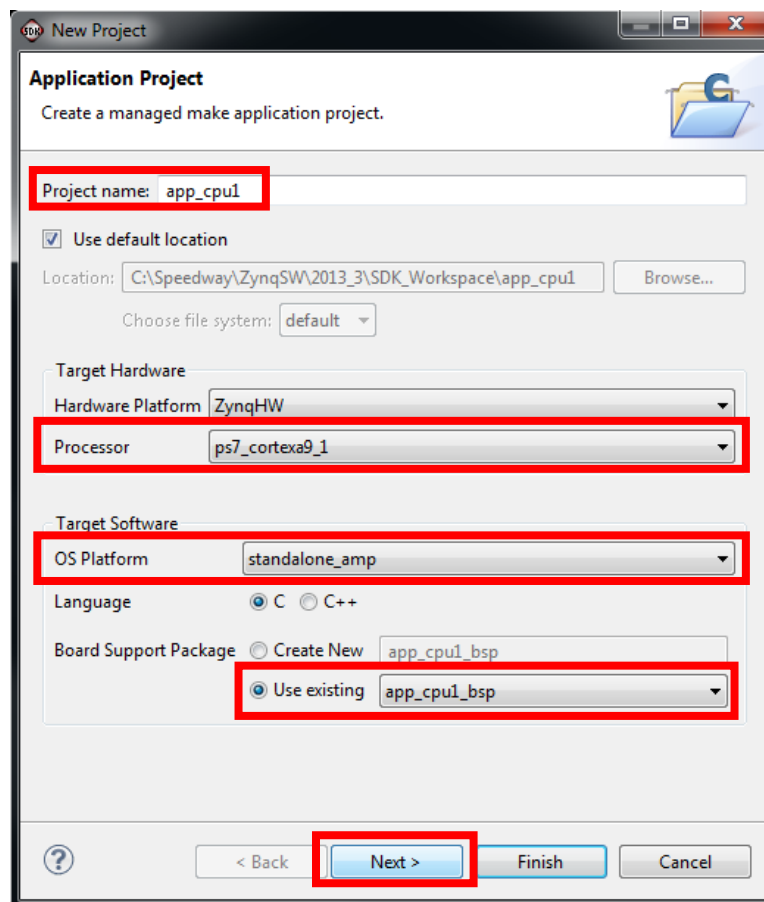
7. In the **New Project** wizard, change the **Project name** field to **app\_cpu1** to match the CPU1 BSP which was created in Step 4.

Use the drop down menu to change the **Processor** setting to the **ps7\_cortexa9\_1** option.

Use the drop down menu to change the **OS Platform** setting to the **standalone\_amp** option.

Change the **Board Support Package** to the **Use existing** option and use the drop down menu to select the existing BSP **app\_cpu1\_bsp** option.

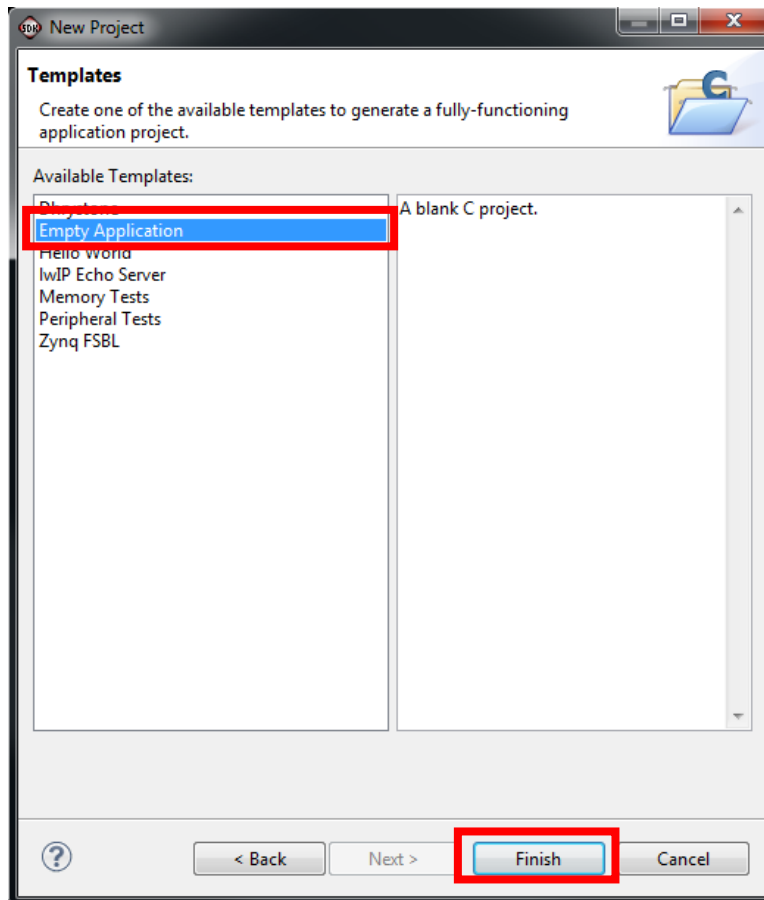
Click the **Next** button to continue.



**Figure 18 – Creating the CPU1 Application**



8. Select the **Empty Application** project template and click the **Finish** button to complete the new project creation process using the **Empty Application** project template.

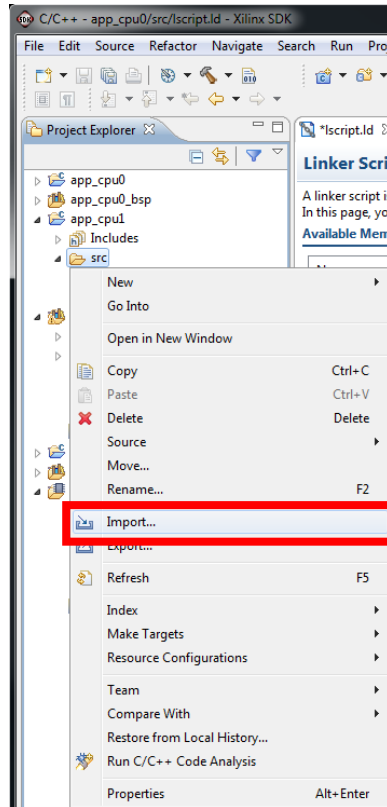


**Figure 19 – Using the Empty Application Project Template**

9. The empty **app\_cpu1** application project is created.

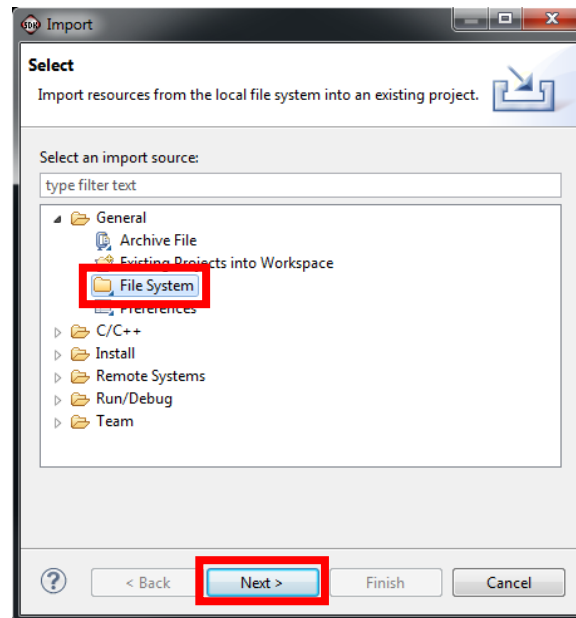
In the **Project Explorer** tab, expand **app\_cpu1** and right-click on the **src** folder.

Click on the **Import** option in the pop up menu.



**Figure 20 – Import CPU1 Application Code**

10. In the **Import** window, expand the **General** item, select the **File System** option, and click the **Next** button.



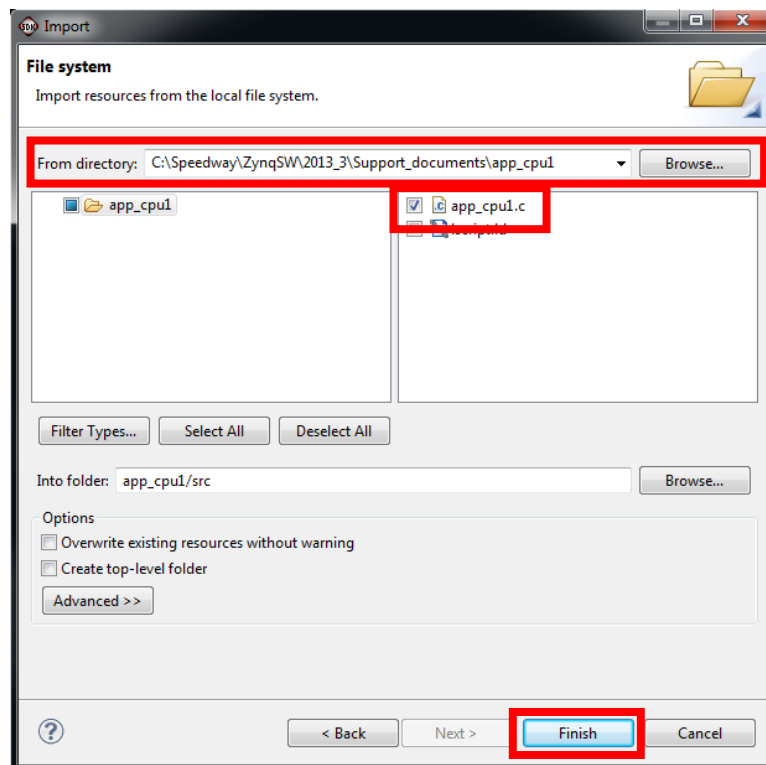
**Figure 21 – Importing from a File System**

11. Click on the **Browse** button and select the following folder which contains the application code that we wish to run on CPU1:

**C:\Speedway\ZynqSW\2013\_3\Support\_documents\app\_cpu1\**

After this folder is selected within the **Browse** dialog, click the **OK** button to search the folder for files to import into the application project.

Select only the **app\_cpu1.c** source file by checking the box next to the file name and then click the **Finish** button to complete the **Import** operation.



**Figure 22 – Selecting Application Source Files**

12. Open the existing **app\_cpu1** project Linker Script **lscript.ld** file. This project was generated automatically when the empty application project was created. The entire memory space from 0x00100000 to 0x1FFFFFFF is assigned to this application which is not only more than is needed for our AMP application, but we also need to reserve a separate memory space for the CPU0 application to execute out of also.

Also, if you were to look at the **app\_cpu0.c** source code you will find that the CPU1 application memory space is expected to start at 0x00200000 so that memory location should become the start of the region for **app\_cpu1**. There is another shared memory region which starts at address 0x03000000 so unless source code modifications are made, we will want to avoid using that memory area when loading the CPU1 application code.

Modify the DDR memory region definition within **lscript.ld** by setting the base address for that region to **0x00200000** and the size of the region to **0x00100000** such that **app\_cpu1.elf** occupies the memory space from 0x00200000 to 0x002FFFFFFF instead of the entire DDR memory space.

**IMPORTANT NOTE:** Be sure to double check that the above address modification is performed. Otherwise, when CPU1 application is loaded into memory by the FSBL, the loaded CPU0 application will be overwritten with the CPU1 application. Once Zynq boots into this condition, this will have the effect of hanging the CPU0 core execution while CPU1 will sleep waiting for an event from CPU0 which will never happen.

Save the changes to **lscript.ld** to rebuild the **app\_cpu1** project with the new memory space definition.

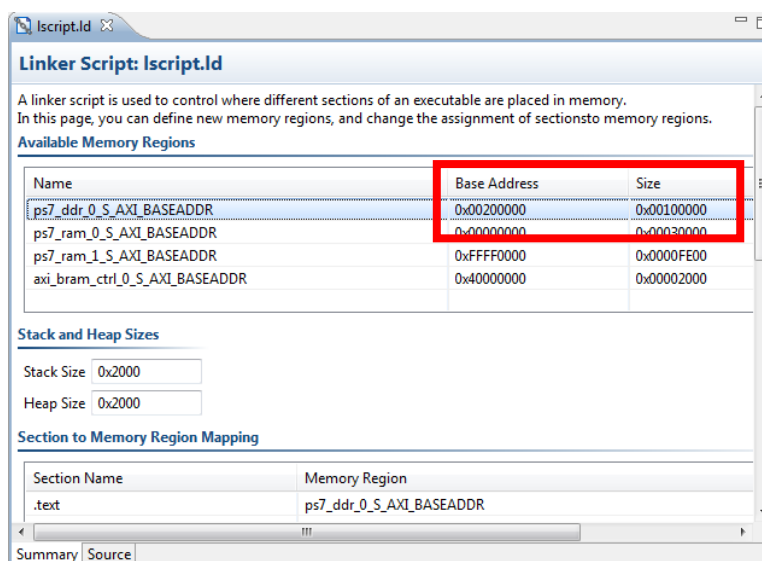
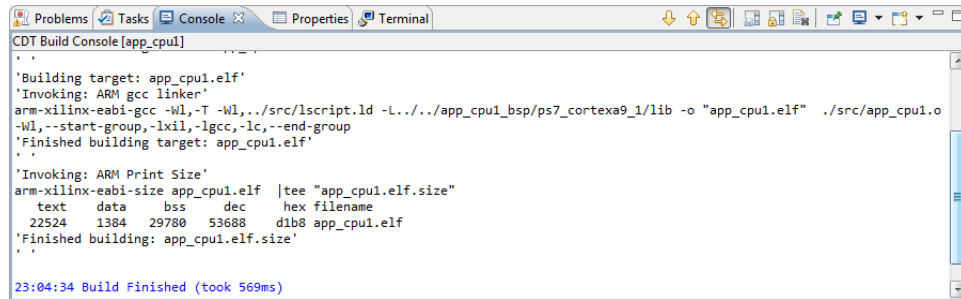


Figure 23 – Modifying lscript.ld

13. The SDK Console panel shows the results of the build. Make sure that the application is built without errors.



```
CDT Build Console [app_cpu1]

'Building target: app_cpu1.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T ./src/lscript.ld -L../app_cpu1_bsp/ps7_cortexa9_1/lib -o "app_cpu1.elf" ./src/app_cpu1.o
-Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: app_cpu1.elf'

'Invoking: ARM Print Size'
arm-xilinx-eabi-size app_cpu1.elf |tee "app_cpu1.elf.size"
  text  data   bss   dec   hex filename
 22524  1384  29780  53688 d1b8 app_cpu1.elf
'Finished building: app_cpu1.elf.size'

23:04:34 Build Finished (took 569ms)
```

**Figure 24 – Application Build Console Window**

14. After SDK finishes compiling the new application code, the ELF is available in the following location:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\app\_cpu1\Debug\app\_cpu1.elf**

### **Questions:**

**Answer the following questions:**

- *Why are two separate application projects required for AMP applications?*

---

## Experiment 3: Booting the AMP Applications on Zynq

This experiment shows how to boot the two AMP applications to their respective CPU cores on Zynq.

When creating a Zynq boot image which boots AMP applications to each of the CPU cores, the following items from the SDK\_Workspace folder need to be added in the order shown:

FSBL - bootloader - **zynq\_fsbl\_0\Debug\zynq\_fsbl\_0.elf**

PL Bitstream - datafile – **hw\_platform\Z\_system\_wrapper.bit**

CPU0 Application - datafile – **app\_cpu0\Debug\app\_cpu0.elf**

CPU1 Application - datafile – **app\_cpu1\Debug\app\_cpu1.elf**

Each of these files will be built into a Zynq boot image file which will be copied to the SD card and renamed to the **boot.bin** filename. The naming of this file is important since this is the filename that the Zynq BootROM searches for when the mode pins have been configured to boot from the SD card.

The Zynq BootROM then opens the **boot.bin** file and searches for the block of data that has been flagged as the **bootloader** partition. The BootROM loads this FSBL application code into OCM and starts running it.

In turn, the FSBL loads the PL bitstream file, the CPU0 ELF, and the CPU1 ELF from their respective boot partitions. At this point, the FSBL that is running on CPU0 jumps to the execution address of the first application ELF that was loaded after the FSBL. As CPU0 starts to run **app\_cpu0.elf**, it writes the starting address of the CPU1 application (0x00200000) to OCM at 0xFFFFFFF0, then executes the assembly instruction **SEV**, which sets an event that wakes up CPU1.

Before the BootROM started running the FSBL, it wrote a very small set of CPU instructions at 0xFFFFFFF0 and set the CPU1 program counter to this location, effectively “parking” the core at this high instruction memory location. This small application on CPU1 checks the contents of 0xFFFFFFF0 and if it is set to 0, executes the **Wait For Event (WFE)** instruction. Every time an event occurs, CPU1 wakes up and reruns the loop where it checks the contents of 0xFFFFFFF0 again for a non-zero value. As soon as a non-zero value is detected, CPU1 jumps to the address location that was read from 0xFFFFFFF0.

In the case of our AMP application, the value stored in 0xFFFFFFF0 is 0x00200000, which is the starting address of the CPU1 application as defined in the **lscript.ld** linker script for the **app\_cpu1** application.

### Experiment 3 General Instruction:

Create a Zynq Boot Image containing the applications for each of the two CPU cores.

Boot the target board to run the AMP applications.

### Experiment 3 Step-by-Step Instructions:

1. Create a **bootimage** sub-folder under the **app\_cpu0** project by right-clicking on the **app\_cpu0** item in the **Project Explorer** pane and selecting the **New→Folder** selection from the pop-up menu.

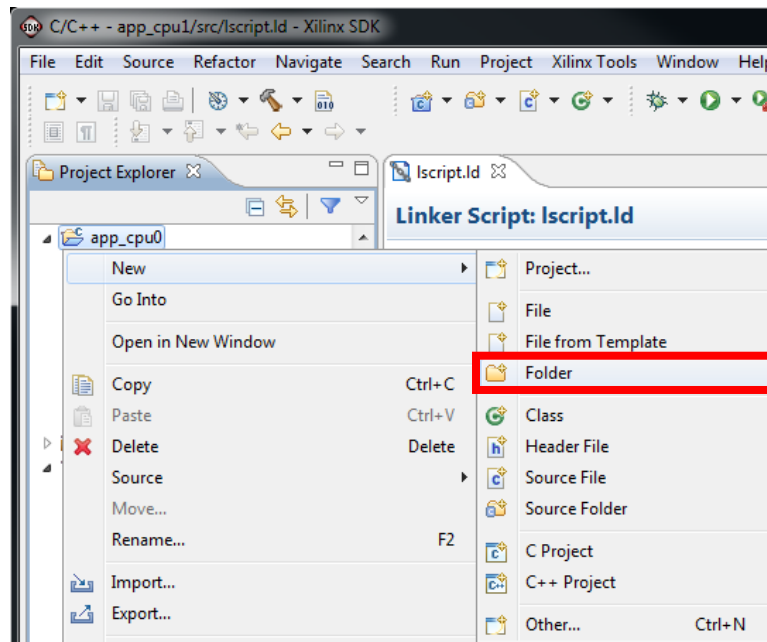


Figure 25 – Creating a bootimage Sub-Folder for app\_cpu0



2. In the **New Folder** dialog, specify the **Folder name** as **bootimage** and then click the **Finish** button. This will create the **bootimage** folder which is where the boot image files will be stored while creating the Zynq boot image in the remaining steps.

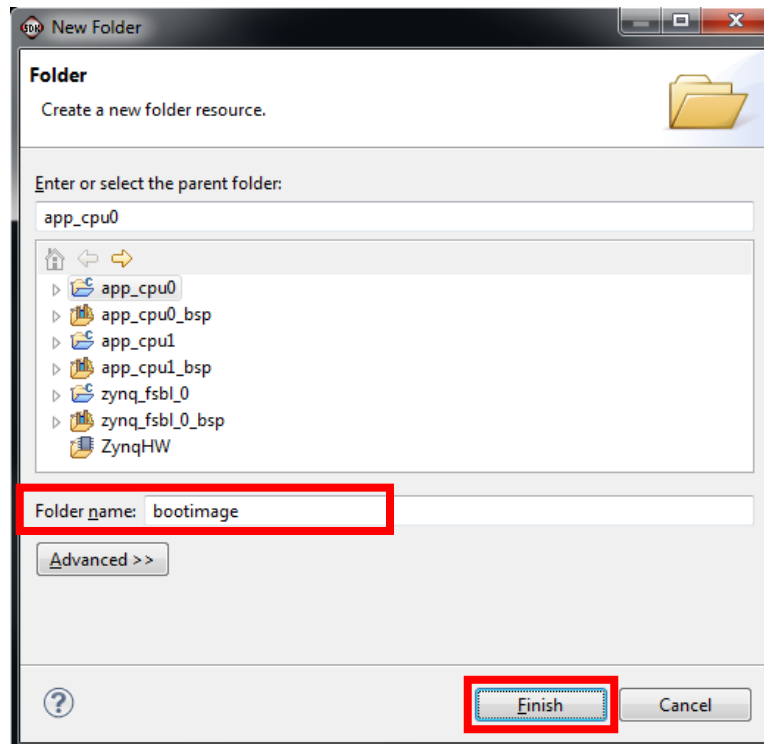


Figure 26 – The bootimage Folder Creation

3. Open the **Create Zynq Boot Image** window by selecting the **Xilinx Tools**→**Create Zynq Boot Image** menu item.

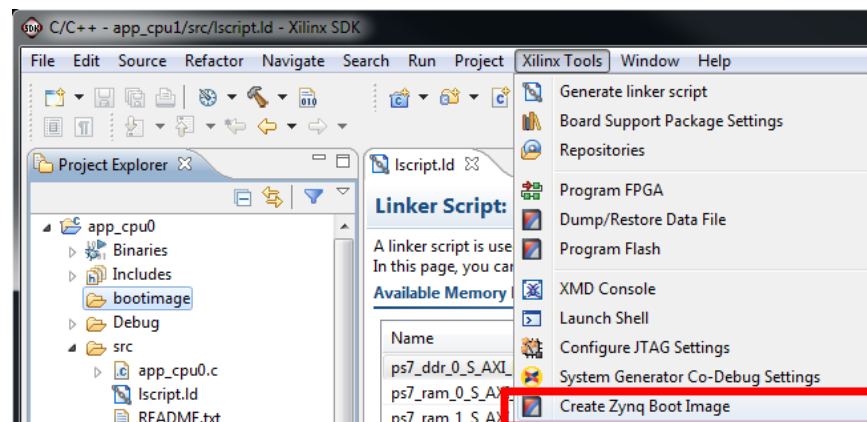


Figure 27 – Opening the Create Zynq Boot Image Tool

4. In the **Create Zynq Boot Image** window, use the **Browse** button next to the **BIF file path** field to create a new Boot Image Format file in the following location:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\app\_cpu0\bootimage\amp\_boot.bif**

For the Boot image partitions list, the following items from the **SDK\_Workspace** folder need to be added in the order shown:

Click the **Add** button then locate the **zynq\_fsbl\_0.elf** and set the **Partition type** of the new partition to **bootloader**. Click the **OK** button to add this new partition to the **Boot image partitions** list.

Click the **Add** button then locate the **Z\_system\_wrapper.bit** and set the **Partition type** of the new partition to **datafile**. Click the **OK** button to add this new partition to the **Boot image partitions** list.

Click the **Add** button then locate the **app\_cpu0.elf** and set the **Partition type** of the new partition to **datafile**. Click the **OK** button to add this new partition to the **Boot image partitions** list.

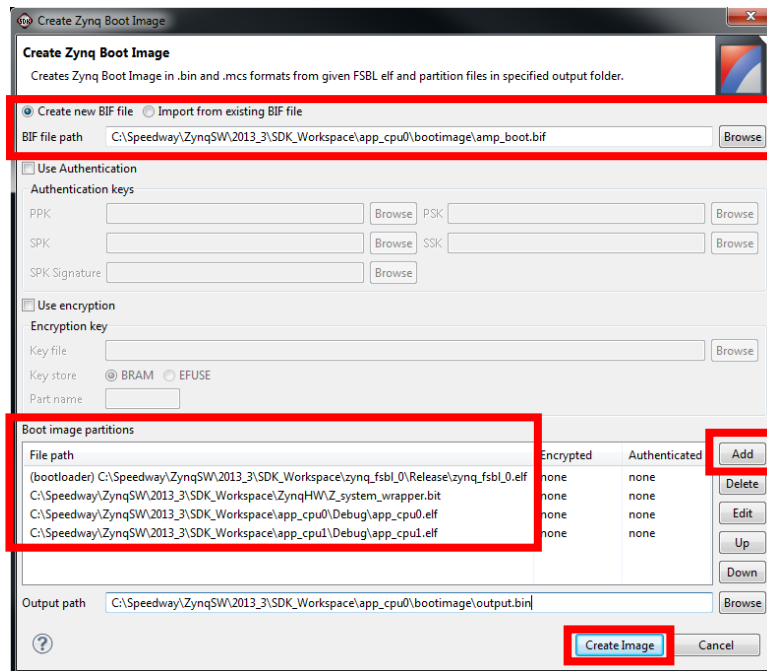
Click the **Add** button then locate the **app\_cpu1.elf** and set the **Partition type** of the new partition to **datafile**. Click the **OK** button to add this new partition to the **Boot image partitions** list.

Files are added in the order shown since this is the order in which their boot image partitions will be processed by the FSBL. Once the FSBL begins processing boot image partitions, it will first encounter the PL bitstream and use the data contained within that partition to configure the PL via the PCAP interface. The next boot image partition encountered is the CPU0 application which is loaded into DDR memory space according to the definitions of the application's linker script and the application start address is stored for later use. The last boot image partition encountered is the CPU1 application executable which will be loaded into the DDR memory space defined within that application's linker script. The final FSBL step is to jump to the start address for the CPU0 application after which the FSBL will no longer execute until a POR or SRST event occurs.

Specify the output path as the following location:

**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\app\_cpu0\bootimage\output.bin**

Click the **Create Image** button to place the Zynq boot image in the **Output** path.

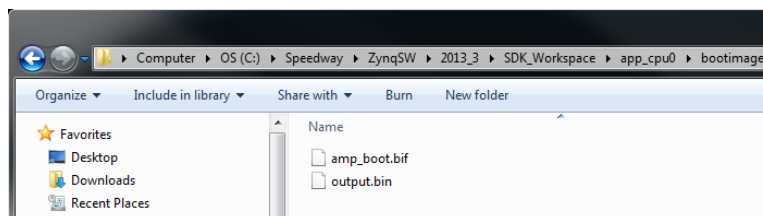


**Figure 28 – Create Zynq Boot Image Tool**

5. Using Windows Explorer, navigate to the following folder:

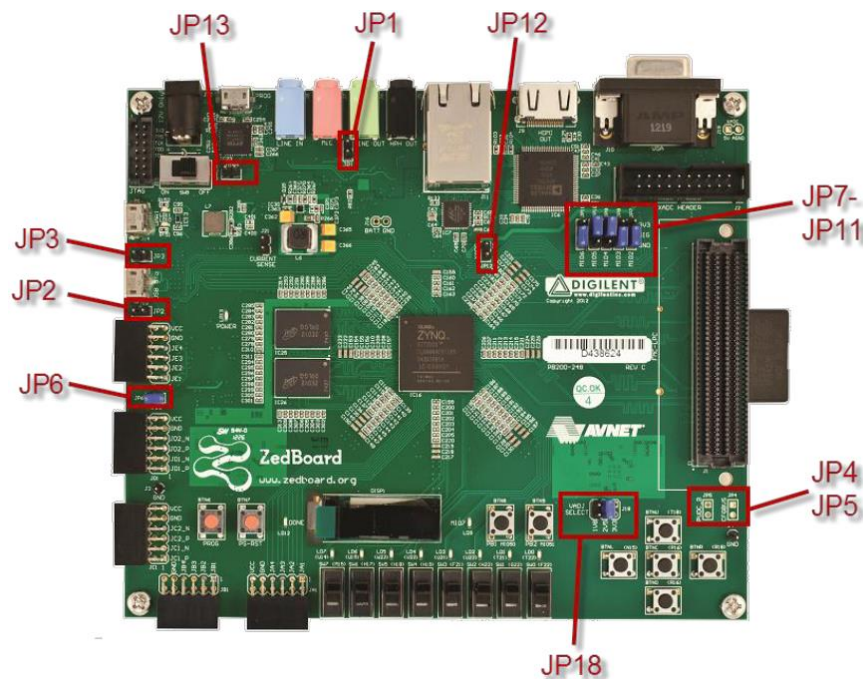
**C:\Speedway\ZynqSW\2013\_3\SDK\_Workspace\app\_cpu0\bootimage\**

Notice that two files have been created: **.bif** and **.bin** files. Only the **output.bin** file is required to boot Zynq from the SD Card.



**Figure 29 – bootimage Folder**

6. Prepare the SD card with the Zynq Boot Image **output.bin** file that SDK generated in Step 4 above.
7. To program an SD Card, insert the SD card into an adapter and plug into your PC. Copy the **output.bin** image to the SD Card.
8. On the SD Card, delete any **boot.bin** files which may be present from earlier lab activities.
9. On the SD Card, rename the **output.bin** file to **boot.bin**. THIS STEP IS CRITICAL!
10. Eject the SD card from the PC.
11. Insert the SD card into the ZedBoard SD card slot (J12).
12. Set the ZedBoard Boot Jumpers to SD Card, with both JP9 and JP10 in the 3V3 position. The jumpers JP7, JP8, and JP11 should be in the GND position.



**Figure 30 – ZedBoard JP7-JP11 Set for SD Card Boot Mode**

13. Turn power switch (SW8) to the ON position. ZedBoard will power on and the Green Power Good LED (LD13) should illuminate.
14. If necessary, launch or re-connect Tera Term.
15. Push the PS-RST (BTN7) button.

16. Observe the terminal output from CPU0 and CPU1 applications.

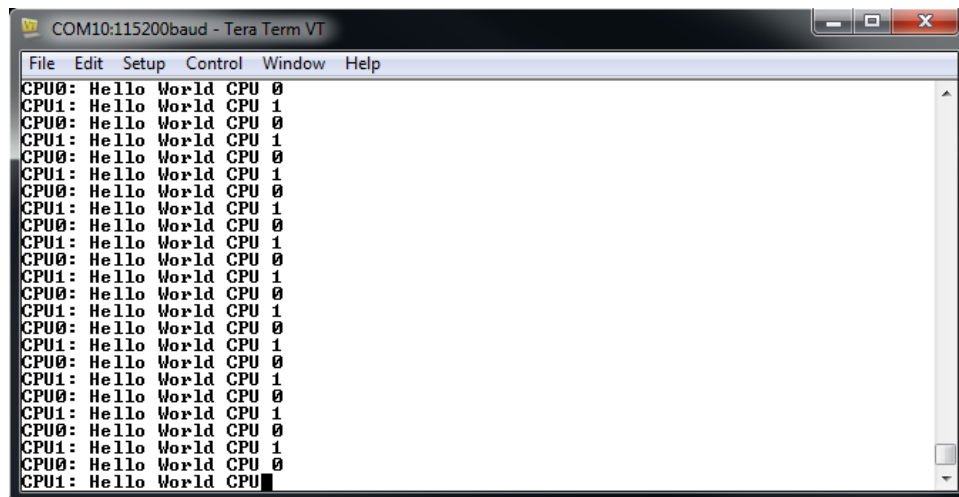


Figure 31 – Terminal Output from CPU0 and CPU1 Applications

### Questions:

**Answer the following questions:**

- What defines the order in which the applications must be loaded to each of the processor cores?

---

## Experiment 4: Debugging AMP Applications on Zynq

This experiment shows how to debug the AMP applications on Zynq.

Xilinx SDK can be used to connect and debug the applications running on both CPUs simultaneously. The TCF debugger provides a hardware "server" that connects to the CPU by way of the JTAG cable. Normally, SDK automatically starts TCF in the background when starting to debug an application. The same technique will be used in this experiment where TCF is automatically connected to both CPU0 and CPU1 targets. Then SDK connects to the TCF session during debug which makes both CPU core execution paths visible to the tools.

Since the FSBL was used to boot the AMP applications, there is no need to reinitialize the PS registers. Care must be taken not to reset the full PS because both CPU applications are to be debugged simultaneously.

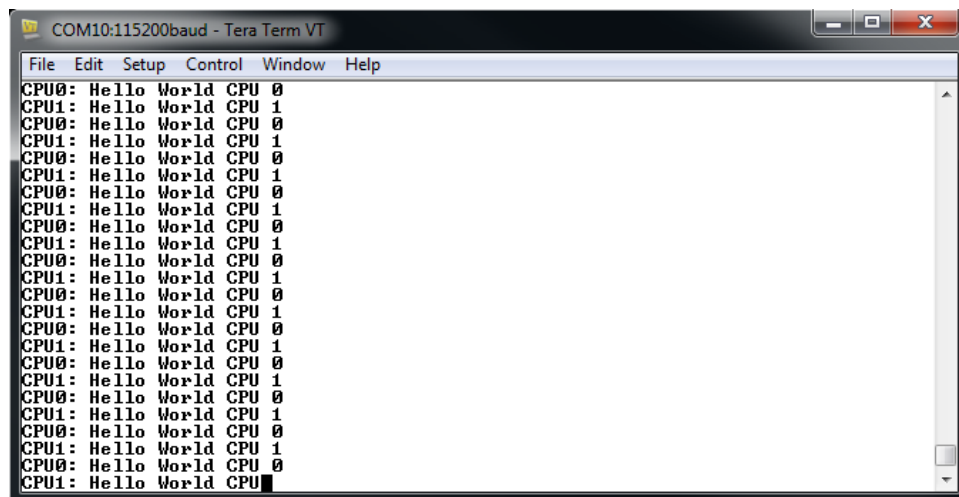
### Experiment 4 General Instruction:

Boot the target board to run the AMP applications.

Connect to the two CPU cores to debug the AMP application code using a TCF debug session.

### Experiment 4 Step-by-Step Instructions:

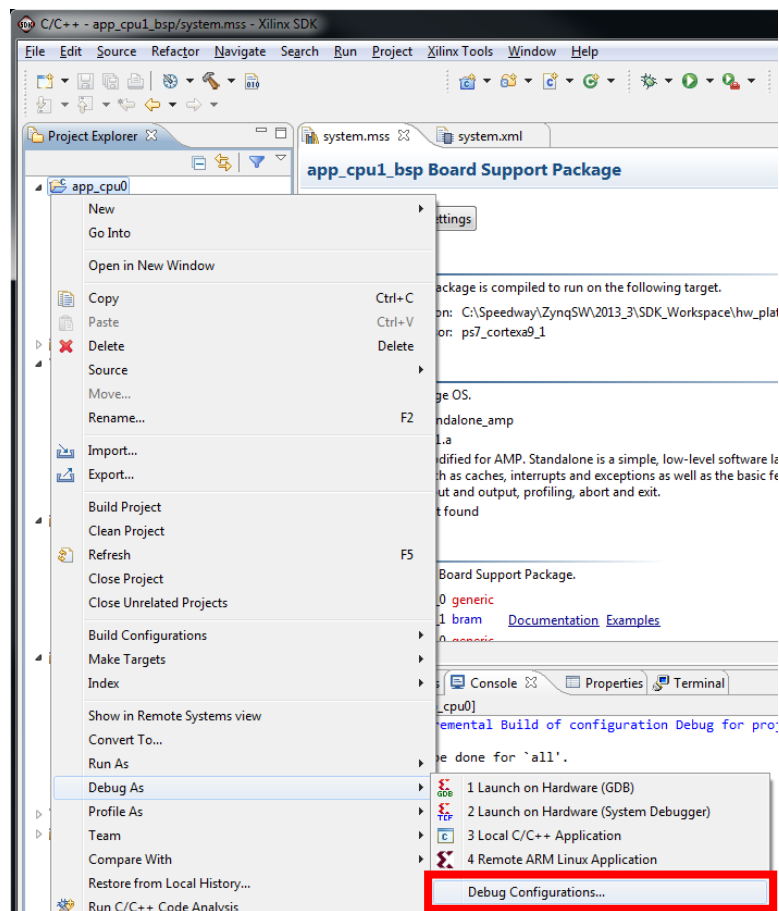
1. Boot ZedBoard to the AMP applications which is where Experiment 3 left off. The console should show a rolling stream of **print()** messages from the two CPU cores.



**Figure 32 – Terminal Output from CPU0 and CPU1 Applications**

2. Attach another Micro-USB cable to the USB-UART port (J17) and make sure that the other end of this cable is connected to the development PC. For this example, the on-board USB-JTAG port will be used however the Digilent HS2 USB-JTAG cable or Xilinx Platform Cable USB II should work in a similar fashion through a connection to the PC4 header J15.
3. In SDK, a new application debug configuration must be created. Debug configurations associate an ELF object file to a target for execution and establish a connection with a debug “server” to perform application debug exercises. In this case, the target is a ZedBoard board accessed over a JTAG connection using the onboard USB-JTAG port.

In the **Project Explorer** tab, right-click the **app\_cpu0** project and select the **Debug As→Debug Configurations...** menu item.



**Figure 33 – Selecting the Application Debug Configurations**

4. From the Create, manage, and run configurations dialog box, double-click the **Xilinx C/C++ application (System Debugger)** option to create a new TCF debug configuration named **app\_cpu0 Debug** by default.

5. Click on the new TCF debug configuration **Application** tab to select it.

Leave the **Download application** option selected in order to load the current application executable into the CPU0 memory so that we can be sure that we are debugging the correct application executable against the current source code. Since each of the CPU cores are already booted by the FSBL, there is no need to use the **Reset processor** option. Also, we should suspend execution by using an implicit breakpoint at the **main()** entry to make it easy to observe the CPU signaling steps at startup.

Deselect the **Reset processor** checkbox option and select the **Stop at program entry** checkbox option.

Verify that **app\_cpu0** is shown as the **Project Name** and that the Application has the **Debug/app\_cpu0.elf** entry.

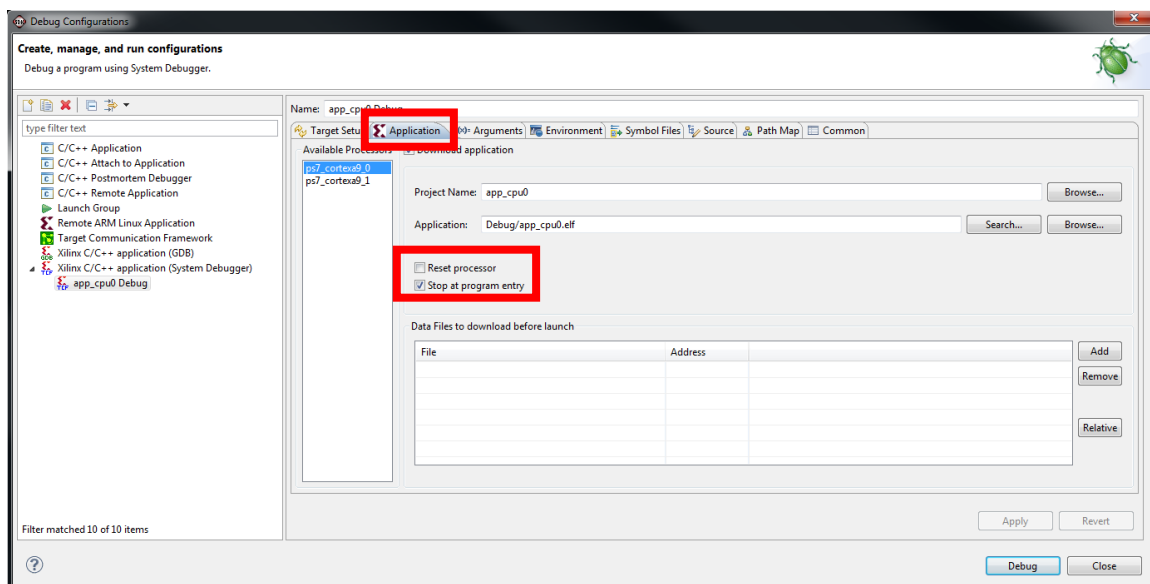


Figure 34 – Processor ps7\_cortexa9\_0 Application Configuration



6. In the same Application tab under the **Available processors** list, select the **ps7\_cortexa9\_1** entry and click the **Download application** checkbox option.

This enables a second application to be specified for CPU1 execution and debug. Deselect the **Reset processor** checkbox option and select the **Stop at program entry** checkbox option.

**IMPORTANT NOTE:** On some smaller resolution screens, the **Debug Configurations** window default sizing does not allow all of these settings to be displayed at once which also can prevent some of the fields from being seen without first either expanding the size of this window or scrolling up within the **Application** tab in order to make the **Download application** checkbox visible on screen.

Verify that **app\_cpu1** is shown as the **Project Name** and that the Application has the **Debug/app\_cpu1.elf** entry.

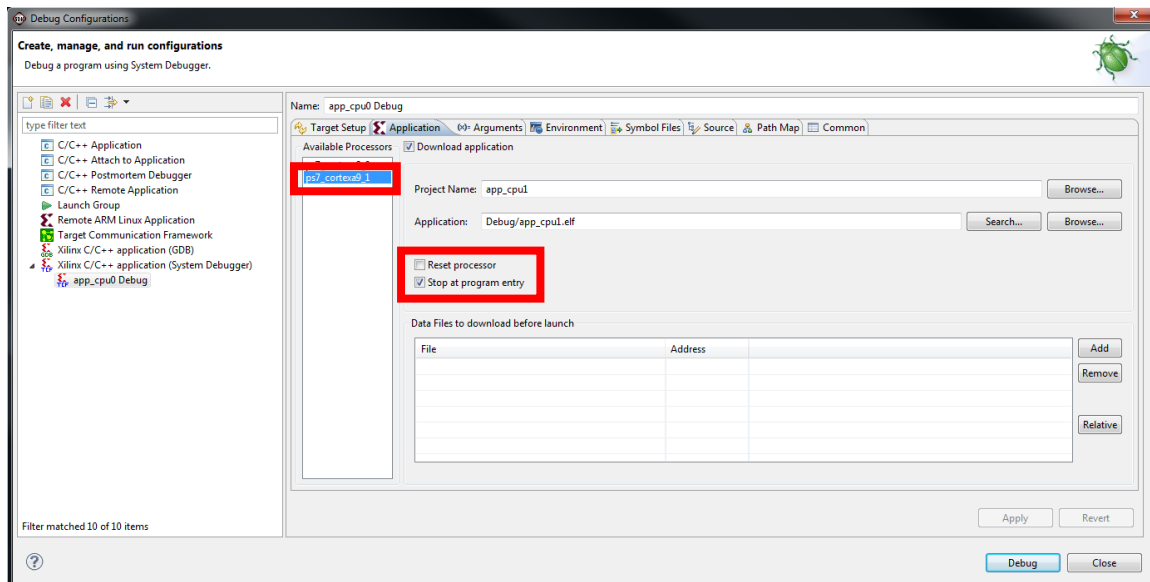


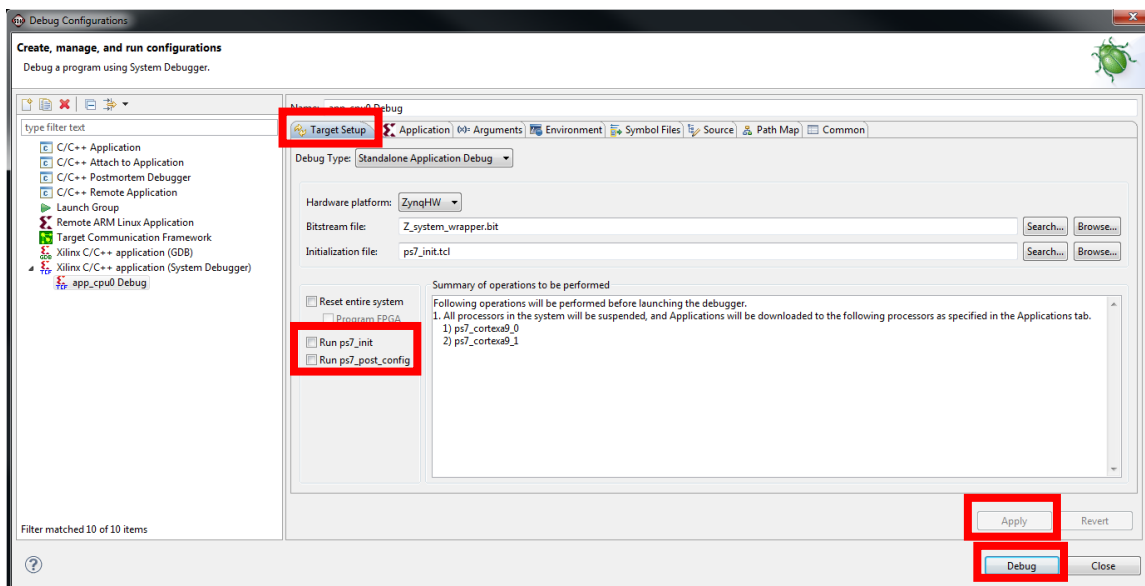
Figure 35 – Processor ps7\_cortexa9\_1 Application Configuration

- Click on the new TCF debug configuration **Target Setup** tab to select it.

Since each of the CPU cores are already booted by the FSBL, there is no need to use the **Run ps7\_init** or **Run ps7\_post\_config** options since this will only re-initialize the CPUs and could potentially undo or alter some of the work already performed by the FSBL.

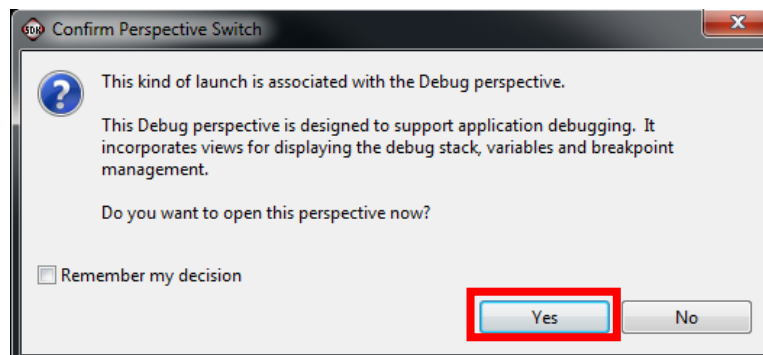
Deselect the **Run ps7\_init** and **Run ps7\_post\_config** checkbox options.

Click the **Apply** button then the **Debug** button to launch the TCF debug session.



**Figure 36 – Target Setup**

- SDK will prompt you to confirm the switch from the **C/C++** development perspective to the **Debug** perspective. Click the **Yes** button to continue launching the debug session.



**Figure 37 – Switch SDK Perspectives**

9. In the TCF **Debug** session window **app\_cpu0 Debug** will be shown and the **APU** hierarchy can be expanded to show application execution on each **ARM Corex-A9 MPCore**.

The applications on core 0 and core 1 should be suspended upon connection. This can be confirmed by checking the terminal output in Tera Term to see that the **print()** messages are no longer scrolling by.

Select the **main()** function listing under the core 0 connection and open the corresponding application source code file **app\_cpu0/src/app\_cpu0.c** by clicking on the stack entry.

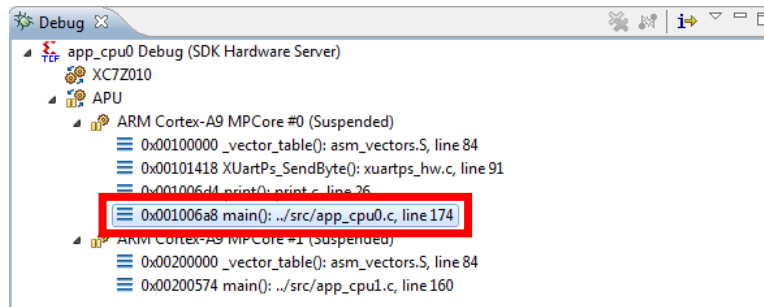


Figure 38 – The app\_cpu0.c Stack Entry

10. Set a breakpoint at line **174** of **app\_cpu0.c** by double clicking within the hatched blue area to the left of the source code.

This will help us to pause application execution once this core has been signaled to access resources shared between the two cores.

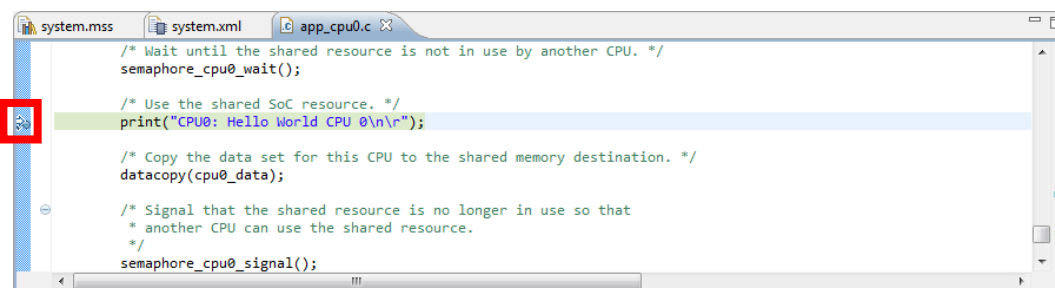
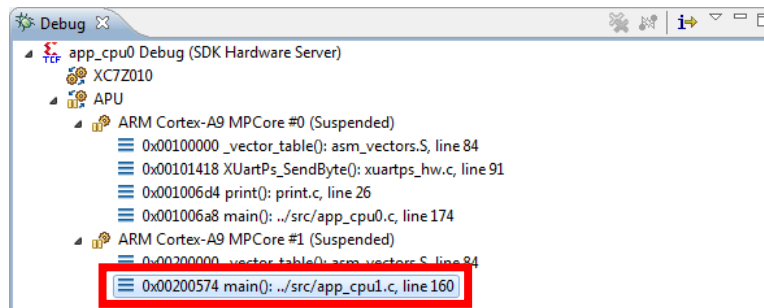


Figure 39 – Breakpoint Set at Line 174 of app\_cpu0.c

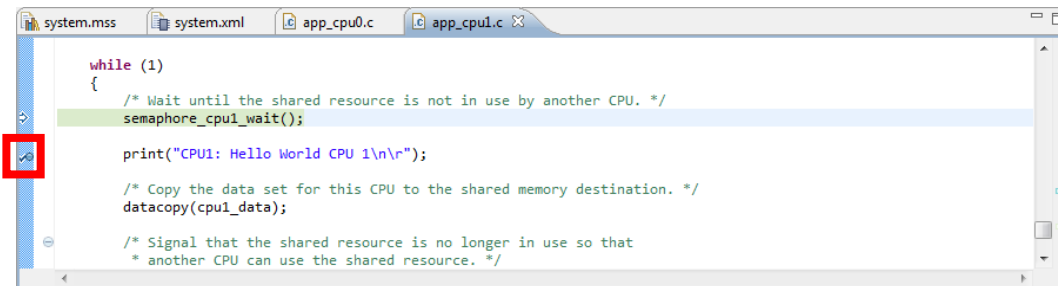
11. In the TCF **Debug** session window, select the **main()** function listing under the core 1 connection and open the corresponding application source code file **app\_cpu0/src/app\_cpu1.c** by clicking on the stack entry.



**Figure 40 – The app\_cpu1.c Stack Entry**

12. Set a breakpoint at line **162** of **app\_cpu1.c** by double clicking within the hatched blue area to the left of the source code.

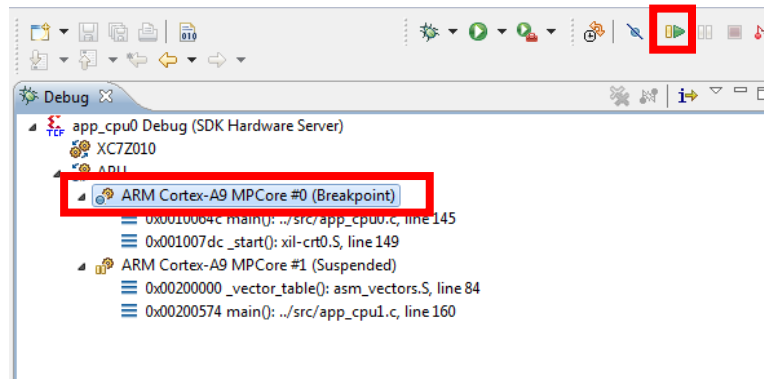
This will help us to pause application execution once this core has been signaled to access resources shared between the two cores.



**Figure 41 – Breakpoint Set at Line 162 of app\_cpu1.c**

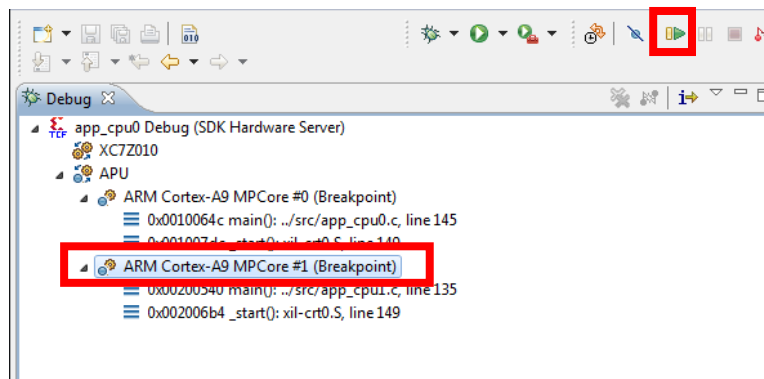
13. Clear the Tera Term buffer so that the application messaging during the startup sequence is not obscured by previous messages. This can be done using the **Edit→Clear buffer** menu option.

14. In the TCF **Debug** session window, select the core 0 connection and then click the **Resume** button to take the application out of the suspended state and advance execution to the implicit breakpoint at the beginning of the **main()** function.



**Figure 42 – app\_cpu0.c at main() Breakpoint**

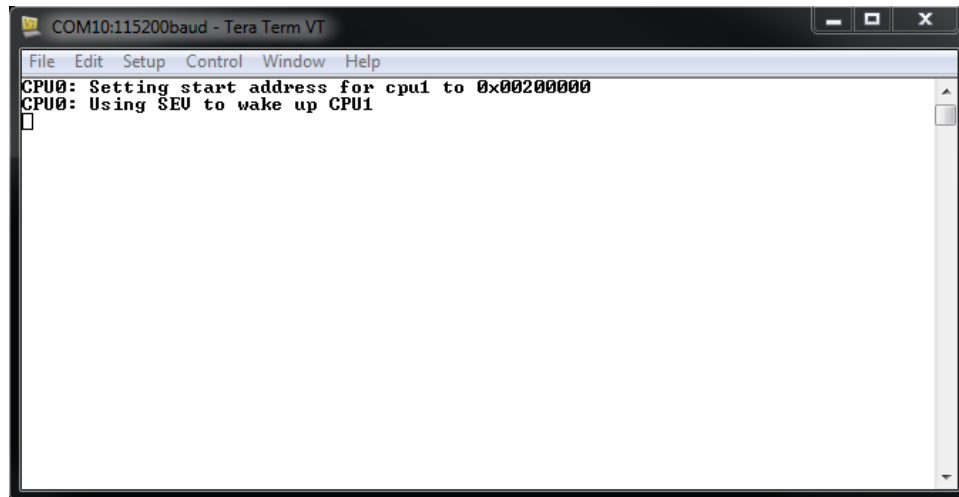
15. Repeat the actions of Step 14 but this time for the core 1 connection.



**Figure 43 – app\_cpu1.c at main() Breakpoint**

16. In the TCF **Debug** session window, select the core 0 connection and then click the **Resume** button again to advance execution to the user breakpoint which was set at line 174 of app\_cpu0.c during Step 10 above.

Notice in the serial terminal that CPU1 execution address has been setup by CPU0 and that the SEV signal has been sent as well.



**Figure 44 – CPU0 Initializing CPU1 Execution Address**

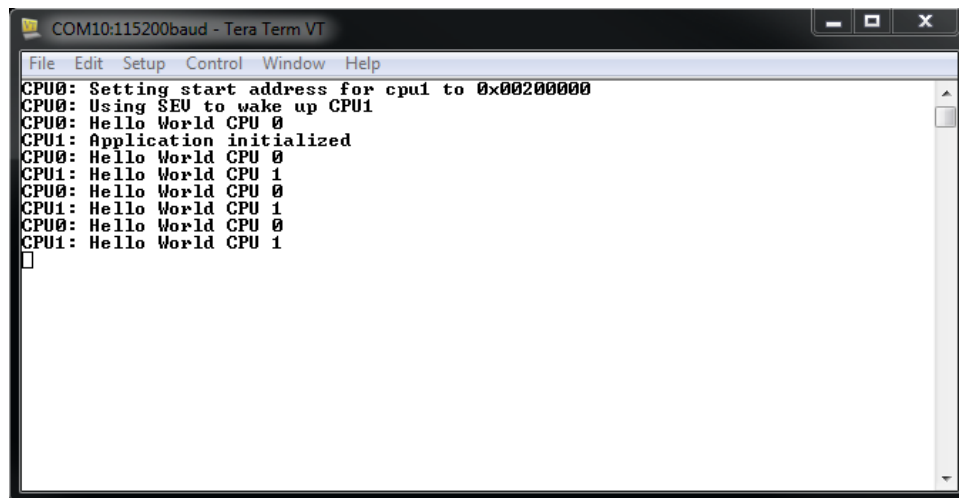
17. Click the **Resume** button once again to start CPU0 into the big **while()** loop and write out the first **Hello World** message. At this point, core 0 has signaled core 1 by writing to the semaphore and will not print a new message until the semaphore is cleared by the core 1 application.

18. In the TCF **Debug** session window, select the core 1 connection and then click the **Resume** button again to advance execution to the user breakpoint which was set at line 162 of app\_cpu1.c during Step 12 above.

Notice in the serial terminal that the CPU1 application has been initialized.

19. Click the **Resume** button repeatedly, notice how each core context is switched in the TCF **Debug** session window depending upon which breakpoint gets hit.

Also notice how a **Hello World** message is printed cleanly to the terminal window depending upon which core execution context is being resumed.

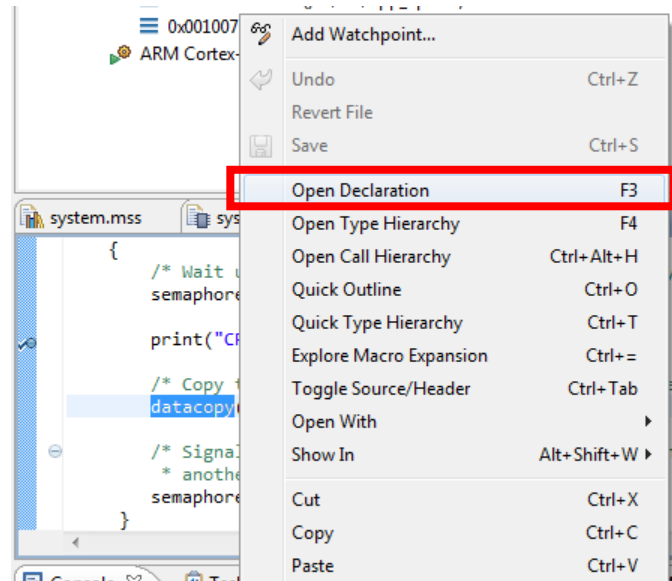


```
COM10:115200baud - Tera Term VT
File Edit Setup Control Window Help
CPU0: Setting start address for cpu1 to 0x00200000
CPU0: Using SEU to wake up CPU1
CPU0: Hello World CPU 0
CPU1: Application initialized
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
```

Figure 45 – Terminal Output from Each Application

20. Evaluate what is being performed within the **datacopy()** functions on each CPU.

The location of the definition of this function is not known without digging around in the source code or searching for the string “datacopy”. One shortcut to the function definition is to highlight the **datacopy** function call name, right-click on the text selection, and then click on the **Open Declaration** menu item.



**Figure 46 – Opening the Declaration for the datacopy() Function**

21. In the **datacopy()** function definition, we can see that some byte data is being copied to a memory location at **DESTINATION\_ARRAY**. Rather than search through code for the **DESTINATION\_ARRAY** definition, simply highlight the definition name, right-click on the text selection, and then click on the Open Declaration menu item.

22. The memory location is again obscured by another definition named **SHARED\_DDR\_MEMORY\_BASE**. Perform the Open Declaration action on this definition as well.



23. The **SHARED\_DDR\_MEMORY\_BASE** definition should be referencing a memory location within the DDR memory space at address **0x03000000**. We can use this address to look into memory to see what sort of data is being copied.

Locate the **Memory** panel within SDK. If the **Memory** panel is not visible within your SDK window, click on the **Window**→**Memory** menu item to display it.

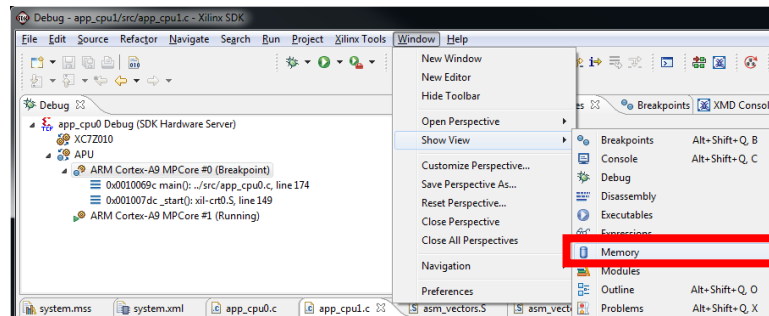


Figure 47 – Displaying the Memory Panel

24. Select the **Memory** panel and click on the Add Memory Monitor button and enter the address **0x03000000** so that the DDR memory contents can be viewed.

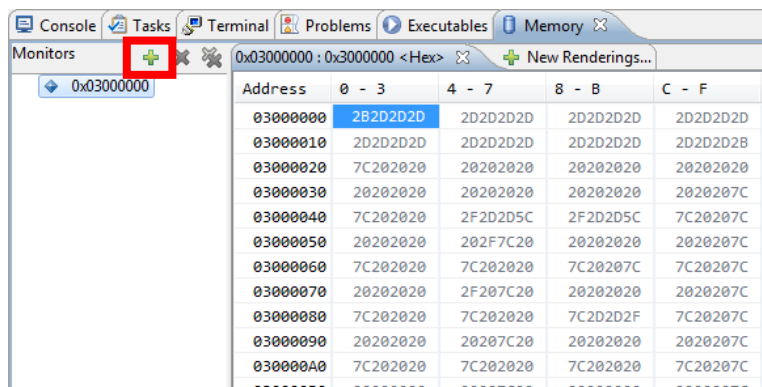
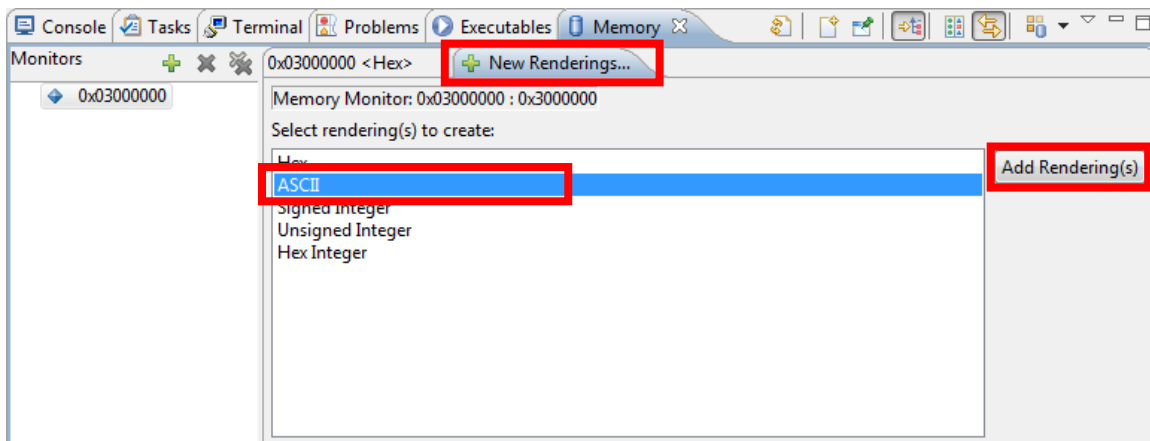


Figure 48 – Hexidecimal Rendered View of 0x03000000 Memory Space

25. This hexadecimal view does not provide us with a lot of information on what is going on in this shared memory space. There are a lot of 0x20 values which correspond to the ASCII white space character so perhaps it would be more useful to view this memory as ASCII characters with a different rendering.

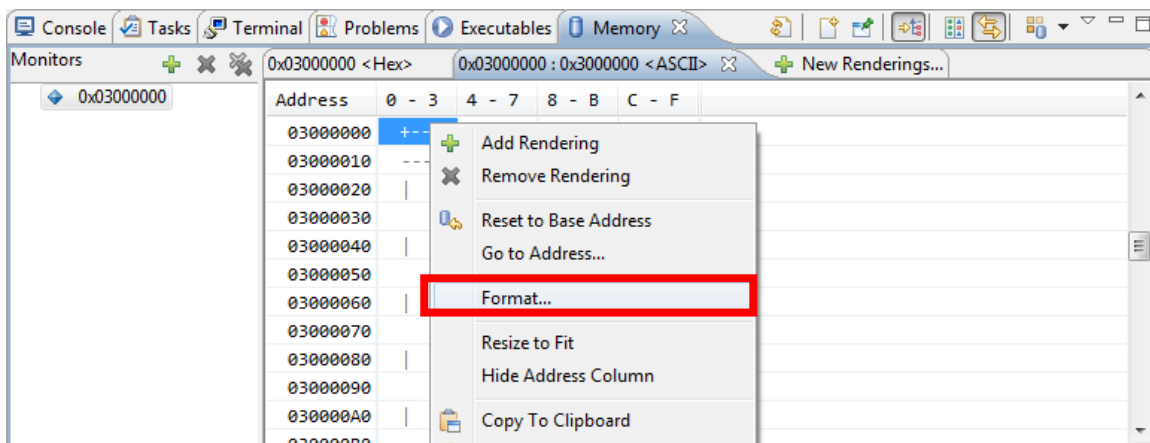
In the **Memory** panel click on the **New Renderings** tab, select the ASCII rendering option, and click the **Add Rendering(s)** button.



**Figure 49 – Creating an ASCII Rendering Memory View**

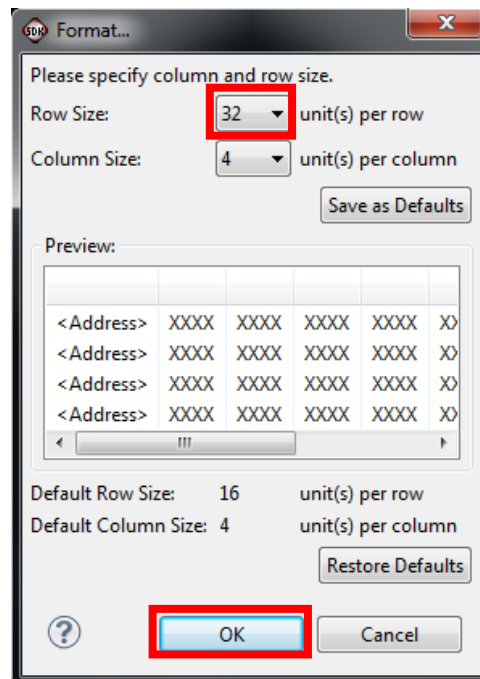
26. Our suspicion of ASCII characters in this memory space appears to be correct, but the contents are not that legible with the current format.

Right click within the ASCII rendered view and select the **Format** option.



**Figure 50 – Adjusting the Format of the ASCII Rendered View**

27. In the **Format** dialog, set the **Row Size** to **32** and then click the **OK** button.



**Figure 51 – Setting the Format Row Size to 32 Units Per Row**

28. Adjusting the Format to 32 units per row appears to be effective at making the ASCII text more legible.

Observe the text pattern within this area of memory. The text displayed is the result of the **datacopy()** function from the last CPU to execute it prior to hitting either a breakpoint or the **semaphore\_cpuX\_wait()** synchronization function.

For the screen shown here, core 0 is stopped at the breakpoint so core 1 executed the **datacopy()** function last.

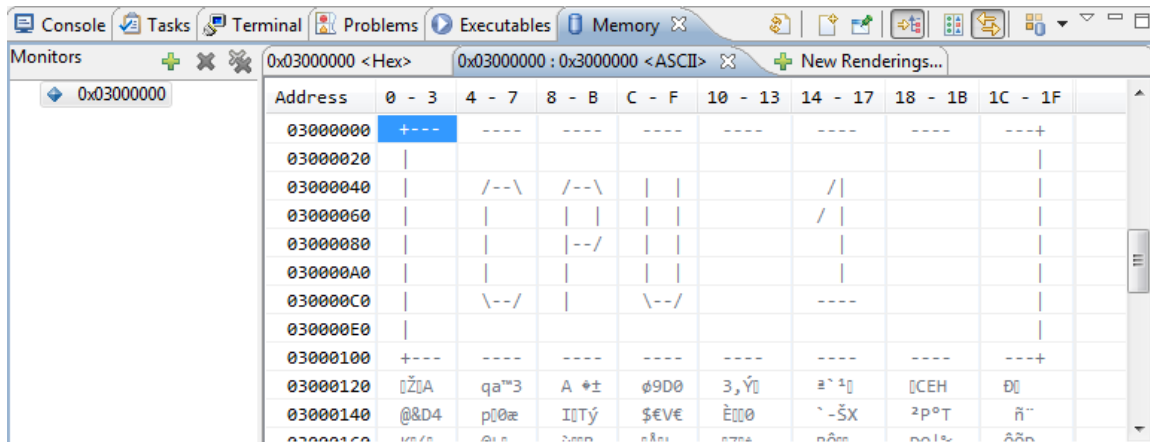
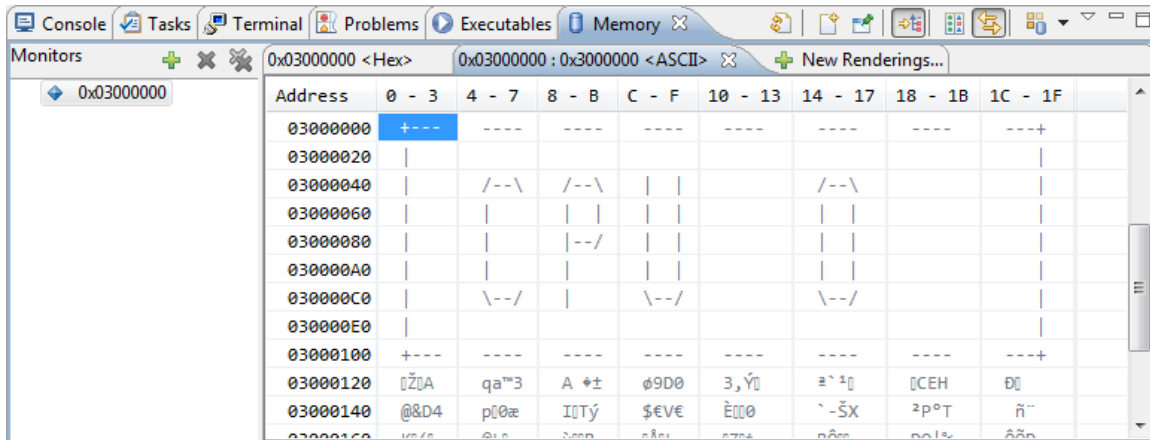


Figure 52 – Adjusting the Format of the ASCII Rendered View

29. Click the **Resume** button once, notice how the memory view blanks when execution context switches to the opposite core after the next breakpoint is reached.

Repeat Steps 24 to 28 above to setup an ASCII rendered view for the other core execution context.

30. Once the ASCII rendered memory view for the other core execution context has been setup, the **datacopy()** results from the other core should be visible.



**Figure 53 – Format of the ASCII Rendered View for the Other Core**

31. Click the **Resume** button repeatedly, notice how each core context is switched in the TCF **Debug** session window depending upon which breakpoint gets hit.

Also notice how the contents of the ASCII rendered memory view change after each time the breakpoint is reached.

32. With execution paused at one of the breakpoints on one core, click on the other core's execution context and watch the memory view. Does the content of the memory view display what you expected?

If you pause execution of the running core by clicking the **Suspend** button, does the content of the memory view now display what you expected?

Notice the highlighted changes that are also marked with the delta  $\Delta$  symbol.

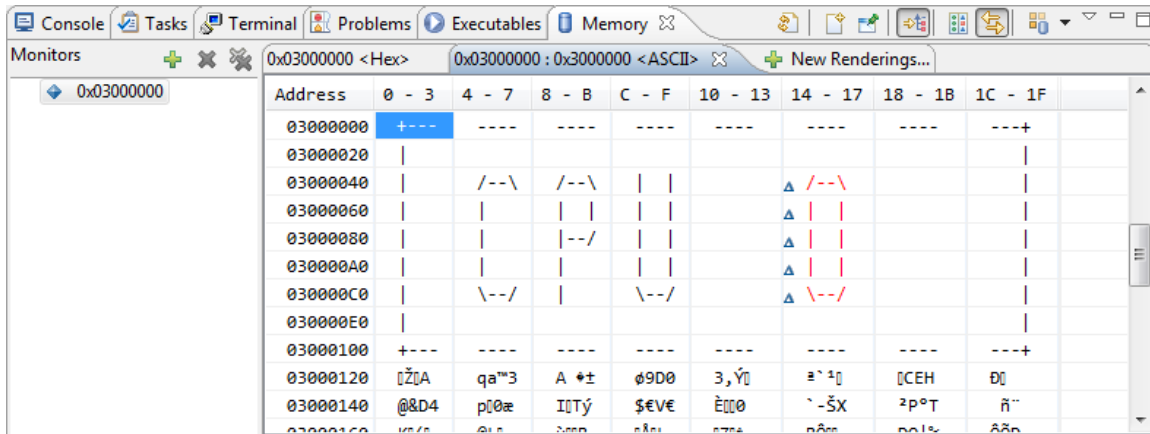


Figure 54 – Format of the ASCII Rendered View after Suspending the Other Core

33. When you are finished with the debug session, click the **Disconnect** button to terminate the TCF debug session.

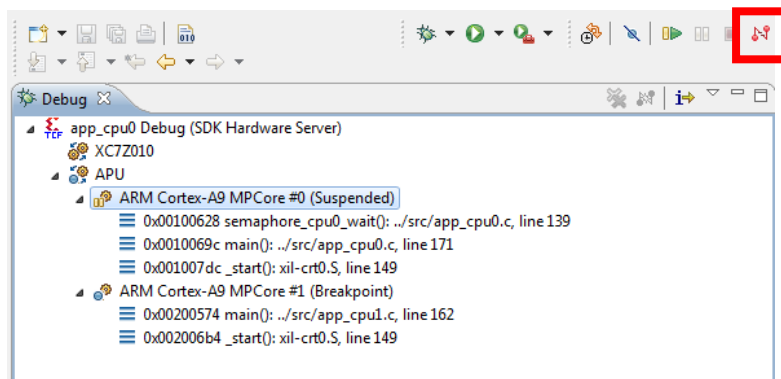


Figure 55 – Disconnecting the TCF Debug Session

## Questions:

**Answer the following questions:**

- *Did the memory view contents display what you expected when execution is paused at one of the breakpoints on one core but you are viewing the execution context of the other core which is still running?*
- 

## Exploring Further

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

- Remove the semaphore calls to **semaphore\_cpuX\_wait()** and **semaphore\_cpuX\_signal()** surrounding the usage of shared resources such as DDR memory and UART. Re-run the applications and discover what application behavior changes.
- Look up the TLB attribute assignments set by the call to **Xil\_SetTlbAttributes()** in UG585 - Zynq Technical Reference Manual.

This concludes Lab 9.

## Revision History

Date	Version	Revision
12 Nov 13	01	Initial release
22 Nov 13	02	Revisions after pilot
01 May 14	03	Correction to memory space in Exercise 2, Step 12 ZedBoard.org Training Course Release

## Answers

### Experiment 1

- *What is the difference between a normal standalone application running on CPU0 and an AMP application running on CPU0?*

There is not necessarily a difference between these two applications other than the necessity of the CPU0 application to manage access to resources shared with CPU1. This can be done through either resource arbitration (such as the DDR) or through inter process signaling (such as a semaphore) in order to avoid contention between the CPUs.

### Experiment 2

- *Why are two separate application projects required for AMP applications?*

Since Zynq shares the DDR memory space between the two CPU cores, each application needs to have its own memory space defined in separate linker scripts and built into the executable at link time.

### Experiment 3

- *What defines the order in which the applications must be loaded to each of the processor cores?*

This is a function of the both the Zynq BootROM and the FSBL. The FSBL is an SDK application for which the Xilinx source code is provided. Since you have access to the source code, you could modify the FSBL to alter the AMP application load order if you should desire this behavior for your Zynq end application.

### Experiment 4

- *Did the memory view contents display what you expected when execution is paused at one of the breakpoints on one core but you are viewing the execution context of the other core which is still running?*

Execution context data, such as memory and register content cannot be updated on the fly with typical debugger technology such TCF. The CPU must be suspended either with a Suspend JTAG instruction or paused at a breakpoint in order for the context information to be updated within the debug session.