Embedded Systems Software Design
Lab Workbook

embd-sw-2016.3-wkb-lab-rev1

Embedded Systems Software Design Lab Workbook 2016.3

Lab FAQ

- Where can I get the files for the labs?
  - www.xilinx.com/training/downloads.htm
  - These are original files and do not contain any work that you may have performed.
  - Labs were developed using version 2016.3 of the tools. Later versions may work and will likely require you to update various pieces of IP. See the "Updating IP" topic in the *Lab Reference Guide* for instructions on how to update IP (Vivado Design Suite Operations > Vivado IP Integrator Operations > Updating IP).
- What if I cannot answer a question in the lab?
  - Do your best! The questions are meant to stimulate thought, not to test your knowledge. After you have pondered a question you can find the answer at the end of each lab.
- Where can I get more detailed information on a topic?
  - The *Lab Reference Guide* is a collection of "how to" topics for commonly performed tasks categorized by tool (Vivado Design Suite, Vivado Analyzer, SDK, etc.) and subdivided into major areas within the tool.
  - The *Lab Reference Guide* is available from the lab files download as well as from www.xilinx.com/training/downloads.htm.
- How do the instructions work?
  - The instructions are provided in layers:
    - Steps – these are the major/broadest aspects of solving the problem that the lab poses (X).
    - Instructions – these represent the significant instructions towards solving the issue outlined by the step (X-X).
    - Tasks – these are the finest granularity items that (when combined with the other tasks) solve the instruction they are subordinate to. Every task is a "clickable" event (X-X-X).

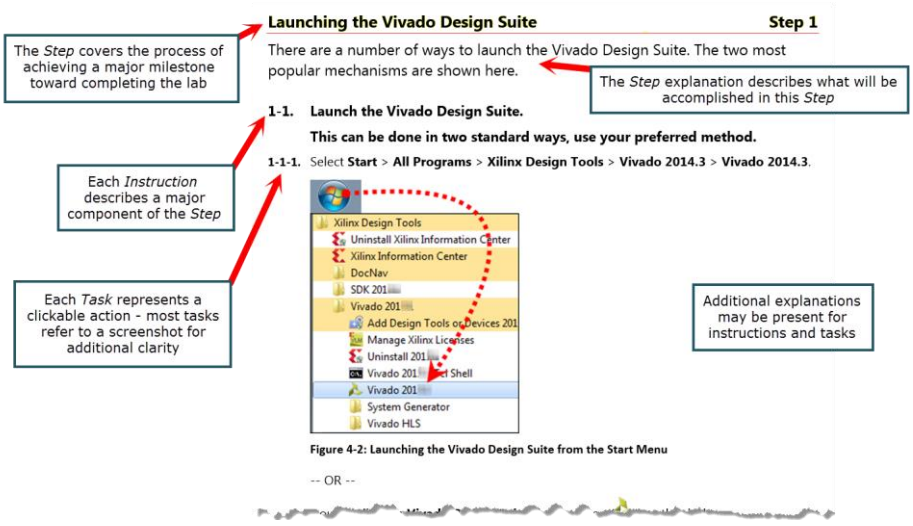


Table of Contents

Lab 1: Exploring the Architecture of the Zynq-7000 All Programmable SoC 3

Lab 2: Exploring the Architecture of the MicroBlaze Processor 17

Lab 3: Driving the SDK Tool 39

Lab 4: System Debugger 66

Lab 5: Application Development 97

Lab 6: File Systems 123

Lab 7: Linker Script 145

Lab 8: Software Interrupts 175

Lab 9: Linux Application Development 197

Lab 10: Boot Loading from Flash Memory 219

Lab 11: SDK Tool Profiling 243

Lab 12: Writing a Device Driver 278

Lab 1: Exploring the Architecture of the Zynq-7000 All Programmable SoC

2016.3

Abstract

This introduction to the basic process of instantiating and customizing the processor system (PS) of the Zynq®-7000 All Programmable SoC family of parts illustrates the process of customizing the PS. While not every aspect of customization is covered, the processes provided here can be extended to all aspects of customization.

Objectives

After completing this lab, you will be able to:

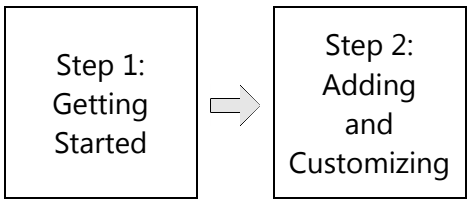
- Instantiate a processor system
- Customize a processor system
- Run the Block Automation tool for a processor system

Introduction

This lab introduces you to the basic process of managing the processing system block in the Zynq-7000 All Programmable SoC family of devices. Included in management is the instantiation of the PS, re-customization, and finally running the Block Automation tool.

The PS is a complex subsystem and an exhaustive investigation of every aspect of this subsystem is beyond the scope of this lab. Only some of the more popular customization are illustrated here with the hope that you will gain familiarity with how these options are organized and that you can quickly find whatever aspect of customization you desire to modify.

General Flow



Getting Started Step 1

As this is a hardware lab focusing on how the Cortex-A9 processor of the Zynq-7000 AP SoC can be configured, all your work will be done with the Vivado Design Suite IP integrator (IPI).

You will begin by launching the Vivado Design Suite, then loading a helper Tcl script to properly configure the project and quickly get you to the IPI tool. If you need to refresh your memory regarding the details of project creation or the mechanics of using the IPI tool, refer to the "Driving the IPI Tool" topic cluster.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. Launch the Vivado Design Suite.

This can be done in two standard ways, use your preferred method.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.3** > **Vivado 2016.3**.

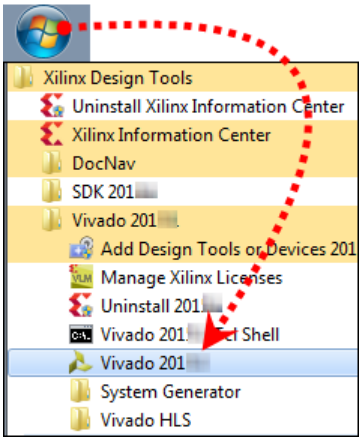


Figure 1-1: Launching the Vivado Design Suite from the Start Menu

-- OR --

Double-click the **Vivado Design Suite** shortcut icon () on the desktop.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

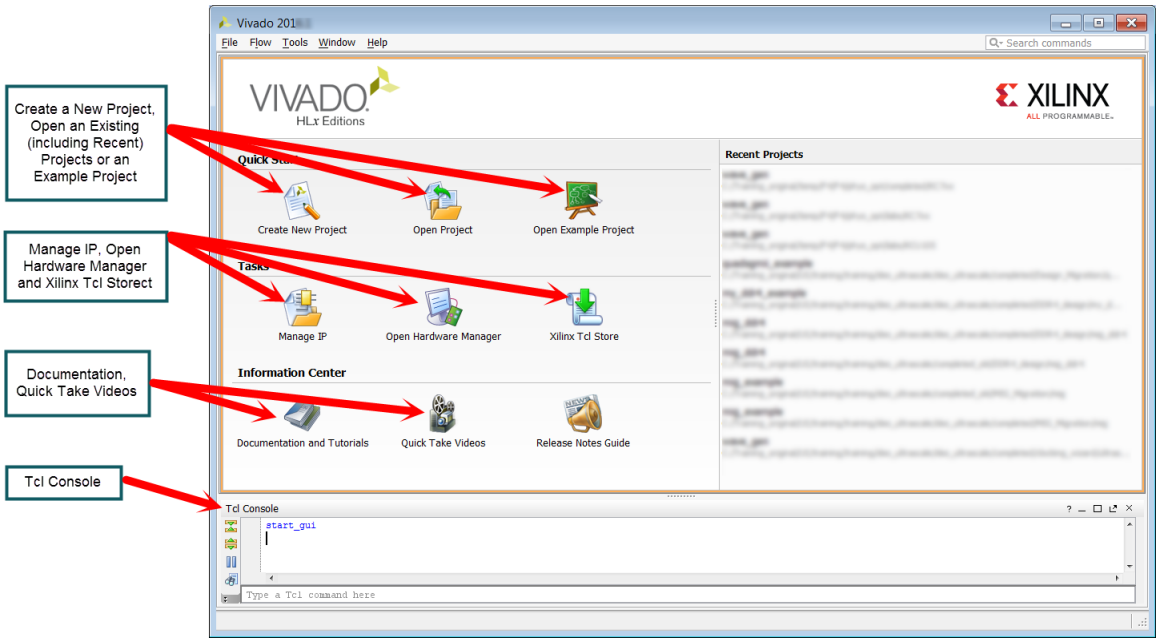


Figure 1-2: Vivado Design Suite Welcome Screen

With the Vivado Design Suite now open, you will load a helper Tcl script and run it to configure the project and get you to the important part of this lab—working with the processor.

The Vivado Design Suite offers both GUI and scripted control. Scripted control takes the form of Tcl commands. These Tcl commands can be entered directly into the tool one at a time, or an entire Tcl script can be loaded and executed.

1-2. Run a Tcl script.

1-2-1. Locate the Tcl command line entry.

The command line entry can be found either on the Welcome page prior to a project being opened, or once a project has been opened.

From the Welcome screen:

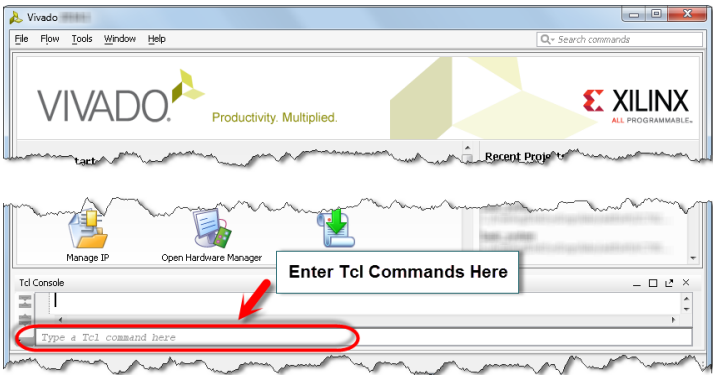


Figure 1-3: Accessing the Tcl Console from the Getting Started Page

From an opened project:

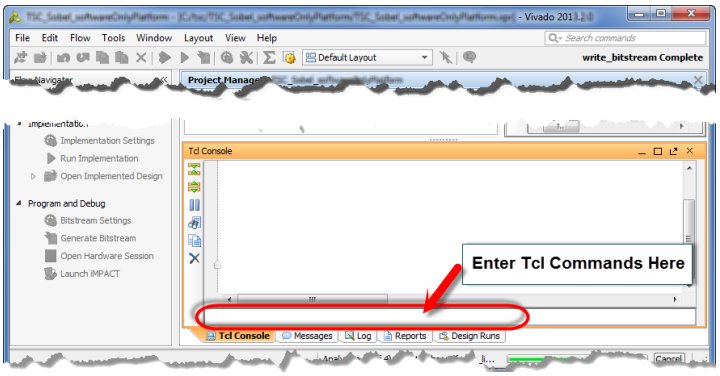


Figure 1-4: Entering Commands into the Tcl Console from an Open Project

The default directory for the Tcl environment is nested within the Xilinx installation directory. This placement, however, is often disadvantageous. In most cases, you will want to navigate to a more useful path. To do this, use the `cd` command to change directory to the user directory.

1-2-2. Change the current working directory to where the Tcl script is located by entering:

```
cd C:\training\ArchZynq7000_Overview\support
```

Remember that the Tcl environment is based on Linux and requires the '/' character to delimit hierarchical paths.

1-2-3. Verify that you are now where you want to be by entering the following into the Tcl command line:

```
pwd
```

The current working directory is displayed. If you are not where you want to be, use the `cd` command to change to `C:\training\ArchZynq7000_Overview\support`.

1-2-4. Enter the following Tcl command:

```
source exploreZynq7000_completer.tcl
```

The Tcl script is run as though you typed each command included in the Tcl script into the Tcl command line. You can follow the execution of the script and monitor for any errors or warnings in the Tcl Console.

With the Tcl script now loaded, you will run the *createProject* and *createBlockDesign* procs to build the Vivado Design Suite project and create an appropriately named block design in which you will perform the remainder of this lab.

1-3. Run the *createProject* and *createBlockDesign* procs.

1-3-1. Enter the following into the Tcl command line to create the Vivado Design Suite project:

```
createProject
```

1-3-2. Enter the following into the Tcl command line to create the block design in which you will build your processor system:

```
createBlockDesign "Zynq7KembdDsgn"
```

Adding and Customizing the Zynq-7000 AP SoC Processor Step 2

With the block diagram canvas open, you will now add and customize a Zynq-7000 AP SoC processor.

2-1. Add a ZYNQ7 Processing System to the IP Integrator canvas.

If you do not recall how to perform this task, refer to the "Adding a Processor to the Block Design" section under IP Integrator Operations in the *Lab Reference Guide*.

Remember that adding a processor IP is just like adding any other type of IP to the canvas. You can also review the "Driving the IPI Tool" topic cluster for a guided example.

Question 1

What information does the wizard provide?

Question 2

How does configuring the PS of the Zynq-7000 AP SoC differ from configuring a MicroBlaze processor?

The Zynq All Programmable PS contains a large number of customizable features. The following instructions will illustrate how to access various sections of the Zynq All Programmable PS, not necessarily indicating which settings to configure.

2-2. Access the Re-customization dialog box.

2-2-1. Double-click the **Zynq PS** icon.

-- OR --

Right-click the **Zynq PS** icon and select **Customize Block**.

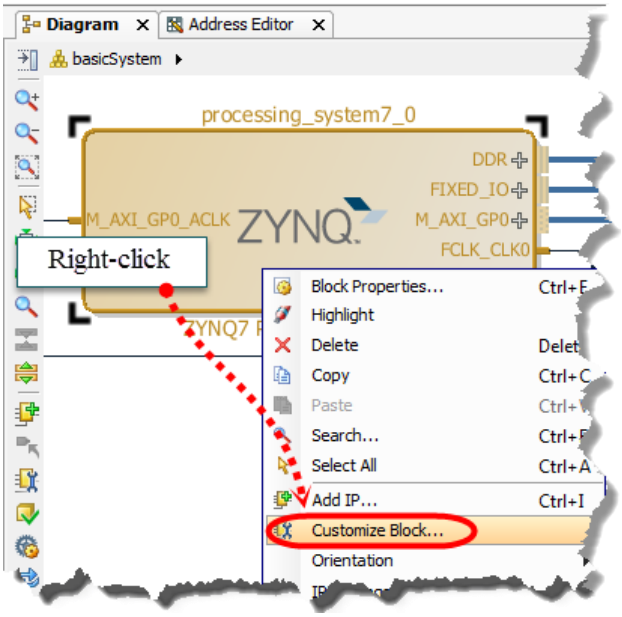


Figure 1-5: Opening the Re-customization Dialog Box for the Zynq All Programmable SoC PS

The Re-customization dialog box opens to reveal the Zynq block design.

2-3. Import board-specific settings.

If you are using an evaluation board, you may want to load the settings for this board as it will contain many of the parameters specific to that board (such as DDR memory timing parameters, enabled peripherals supported by that board, etc.)

You can also create a *.bd* file for your custom board if you want and import those settings.

2-3-1. Click **Presets**.

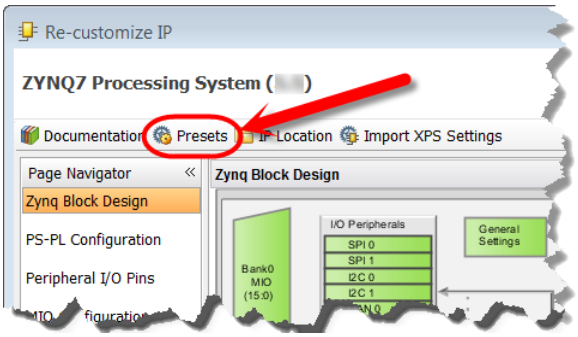


Figure 1-6: Accessing the Presets Button

2-3-2. Select the template for **ZedBoard**.

All the Xilinx boards that support the device that was selected during the project creation are shown under the System Template (Presets) drop-down list. Even if you selected the part by board, you have the opportunity to select a different board that has the same part on it.

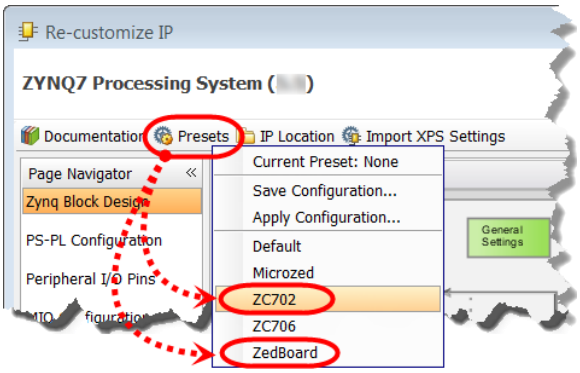


Figure 1-7: Selecting a Preset Template (ZedBoard and ZC702 Shown as Examples)

The next instruction provides you with general guidance as to how to customize various aspects of the PS. At the end of this instruction you will be provided with specific guidance regarding how you are to customize the PS.

2-4. Select the page or specific block to re-customize.

2-4-1. Click the topic in the Page Navigator or any green (active) box from the Zynq Block Diagram.

Along the left are the navigational tabs that you will use to access different groups of parameters.

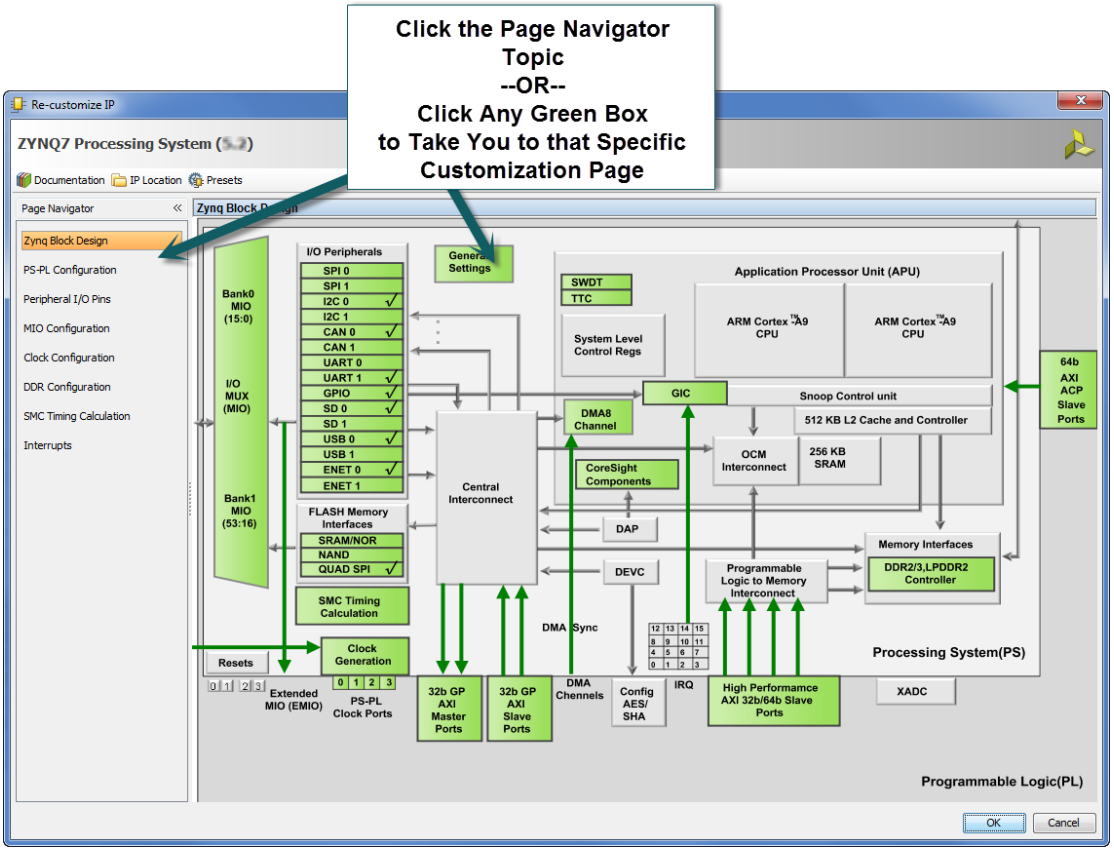


Figure 1-8: Navigating the Zynq PS Recustomization Dialog Box

- PS-PL Configuration: This page contains all the signals that cross the PS-PL boundary. These signals are broken down further into:
  - General: Contains default baud rates for the UARTs, FTM Trace buffer settings, PS-PL cross triggering enables, enables for the clock triggers, and reset.
  - DMA Controller: Contains enables for the four peripheral request interfaces.
  - GP Master AXI Interface: Enables/disables access to the GP Master AXI Interfaces to the PL.
  - GP Slave AXI Interface: Enables/disables access to the GP Slave AXI Interfaces from the PL.
  - HP Slave AXI Interface: Enables/disables access to the GP Slave AXI Interfaces from the PL and sets the port width.
  - ACP Slave AXI Interface: Enables/disables access to the ACP Slave AXI Interface from the PL.

- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

The following is how your system should be configured:

- Use the ZC702 or Zed preset
- Disable the following features:
  - All peripherals except as listed below (accessible through either the Peripheral I/O Pins or the MIO configuration or Block Diagram > I/O Peripherals)
    - UART 1
    - QSPI
- Enable the following features:
  - HP Slave AXI Interface > S_AXI_HP0 interface
  - QSPI (enabled by default through the preset)
  - UART1 (enabled by default through the preset)

2-4-2. Configure the PS according to the list above.

2-4-3. Click **OK** to accept the configuration and return to the IPI canvas.

Remember that you can re-configure the processor at any time. However, use caution as you may need to re-run Designer Assistance to handle any changes you make.

Many IP blocks are supported by the Designer Assistance feature for automating the configuration of an IP block as well as block-specific connections. This is referred to as Block Automation and allows designers to quickly configure new IP blocks for common use cases.

2-5. Use Block Automation to automate the configuration of the recently instantiated IP.

2-5-1. Click **Run Block Automation** from the Designer Assistance information bar.

Figure 1-9: Designer Assistance Offering Block Automation

This opens a Run Block Automation dialog box listing all the IP currently in the design eligible for block automation. IPs are listed in a hierarchy on the left and any options associated with a particular automation will be shown in the right pane whenever an IP instance is selected in the left pane.

2-5-2. Click the **Expand All** icon ( ) to ensure that the list of available automations is fully visible.

2-5-3. Select either the top-level **All Automation** check box or individual blocks as listed below. Unless otherwise indicated within the block list, maintain default automation options for all blocks.

- Blocks: processing_system7_0

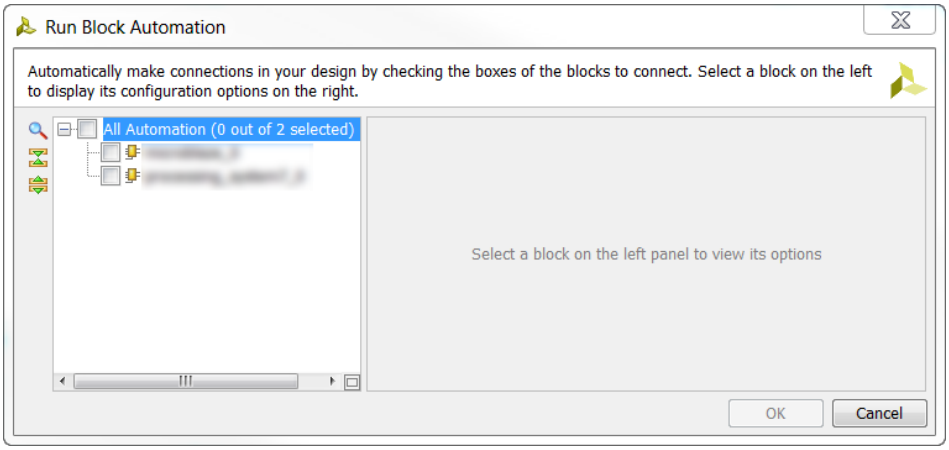


Figure 1-10: Run Block Automation Dialog Box

Note the various options that are selectable for the PS.

Question 3

Do any of these options look familiar?

2-5-4. Deselect the **Apply Board Preset** option to avoid overwriting the current PS settings.

The Cross Trigger In and Out selections are used in co-debugging to enable the processor(s) to cause the debug logic (ILAs) in the PL to trigger and to have the trigger signal generated by the debug logic (ILAs) cause the software to pause (a la breakpoint).

2-5-5. Click **OK** to run the selected automation.

Question 4

What does the Block Automation do?

At this point, the PS has connections to external memory (DDR), has its own internal memory for booting, and has a collection of peripherals available for I/O. As is, this could operate with no further additions.

Question 5

What happens to the unconnected port to the processing system?

Summary

You just completed the basic tasks of instantiating the processing system of a Zynq-7000 AP SoC, configuring it, and using the Block Automation tool to connect the DDR and fixed I/O to the outside world.

Answers

1. What information does the wizard provide?

   The opening page of the wizard provides two methods for locating the aspect of the processing system that can be modified. The Page Navigator (on the left side of the wizard) provides a textual list of the various configurable areas, whereas the Zynq Block Design graphic enables you to quickly locate a configurable aspect by clicking any of the green blocks.

   Along the top menu, you can retrieve documentation for this processor, load a preset (pre-configurations for various boards including DDR settings, Flash configuration, etc.), and other less frequently used options.

2. How does configuring the PS of the Zynq-7000 AP SoC differ from configuring a MicroBlaze processor?

   The MicroBlaze processor is a softcore processor and the various configurations will impact overall performance, the number and type of resources used, and maximum clock frequency.

   The Zynq AP SoC's processing system is dedicated silicon and, while peripherals, clocks, and some connections can be configured, the maximum frequency, performance, and consumed resources is fixed and cannot be changed.

3. Do any of these options look familiar?

   The Apply Board Preset option is a duplicate of the process that you performed inside the Re-customization window. If you leave the Apply Board Preset option selected at this stage, it will re-configure the PS to the default board settings and you will have to re-customize the PS again.

4. What does the Block Automation do?

   The Block Automation makes the connections to the DDR and the fixed I/O. These are the dedicated pins of the PS. Because they are dedicated, they do not require any constraints.

   Remember that the fixed I/O is controlled via the MUI and can be dynamically reconfigured.

5. What happens to the unconnected port to the processing system?

   The unused ports (even though they were defined) connect between the PS and PL. If there is no logic in the PL, then these ports simply *dangle* and are not used.

   One should be careful, though, not to drive AXI transactions out the master AXI general-purpose port because, since there is no *receiving* AXI logic, the port will hang and likely crash the system.

Lab 2: Exploring the Architecture of the MicroBlaze Processor

2016.3

Abstract

Some of the configurable options in the MicroBlaze™ processor are introduced in this lab. The student will learn how to instantiate and configure the MicroBlaze processor and use Designer Assistance to complete a design.

Objectives

After completing this lab, you will be able to:

- Instantiate the MicroBlaze processor
- Customize the MicroBlaze processor
- Invoke the Design Automation tools to add support logic to the design

Introduction

Once you have completed the "Driving the IPI Tool" topic cluster (which illustrates how the Vivado® IP integrator tool is used to create and manipulate IP blocks), you can begin to create a MicroBlaze processor design.

Here you will instantiate a MicroBlaze processor and explore different customizations, both predefined and user specific. Once the MicroBlaze processor is configured, you will run the Block Automation tool to add in optional supporting logic, such as clocks and resets, interrupts, memory, etc. Next you will use the Connection Automation tool to connect the remaining pieces of IP. Finally, you will add a UART peripheral and connect it to the system.

The following figure shows the high-level view of the design that you will build.

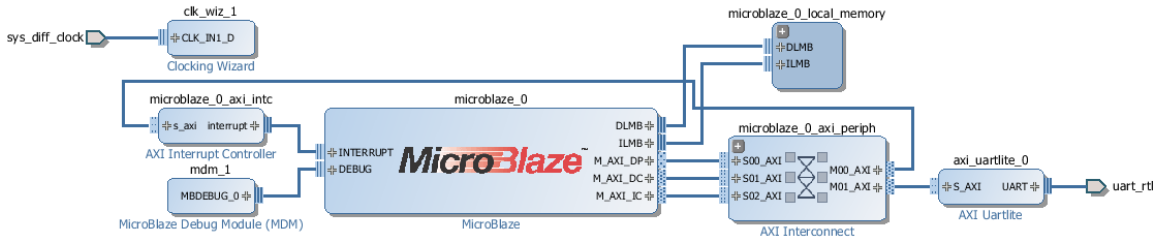
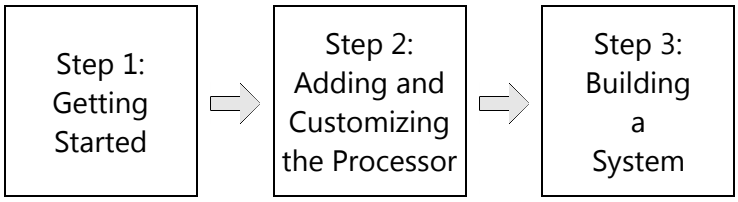


Figure 2-1: Block Diagram of the MicroBlaze Processor Design for this Lab

General Flow



Getting Started Step 1

As this is a hardware lab focusing on how the MicroBlaze processor can be configured, all your work will be done with the Vivado Design Suite IP integrator (IPI).

You will begin by launching the Vivado Design Suite, then loading a helper Tcl script to properly configure the project and quickly get you to the the IPI tool. If you need to refresh your memory regarding the details of project creation or the mechanics of using the IPI tool, refer to the "Driving the IPI Tool" topic cluster.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. Launch the Vivado Design Suite.

This can be done in two standard ways, use your preferred method.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.3** > **Vivado 2016.3**.

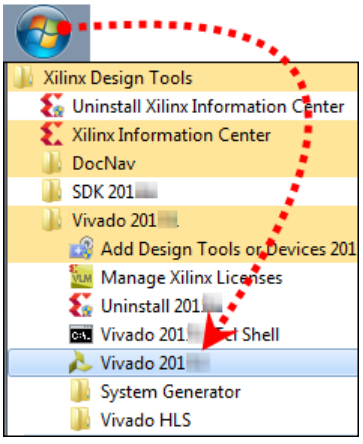


Figure 2-2: Launching the Vivado Design Suite from the Start Menu

-- OR --

Double-click the **Vivado Design Suite** shortcut icon () on the desktop.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

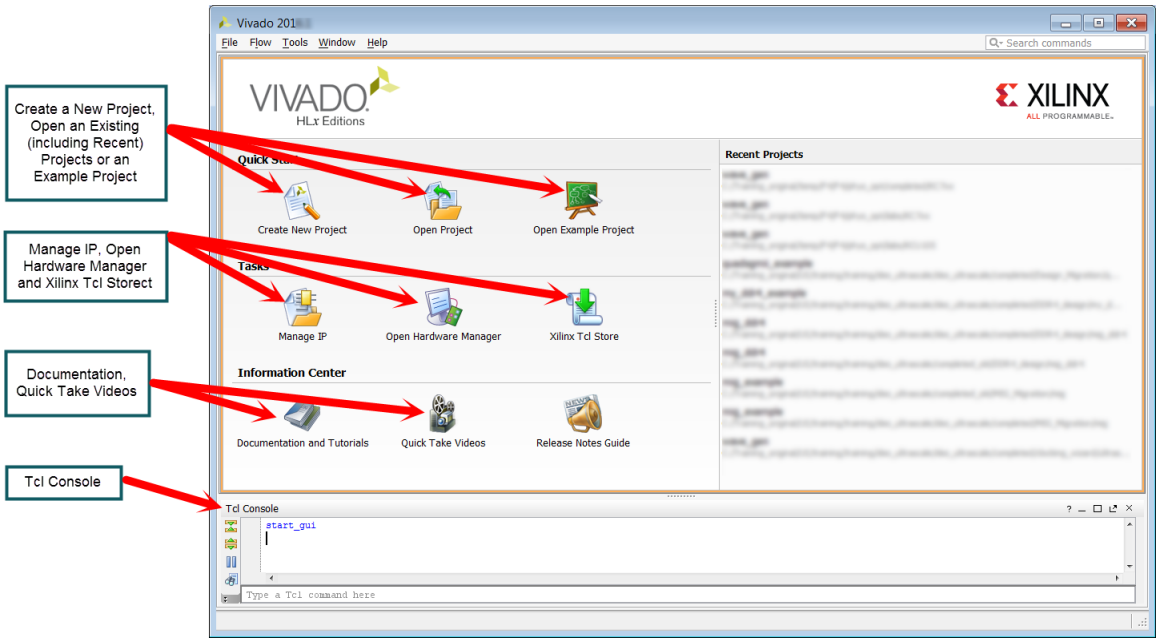


Figure 2-3: Vivado Design Suite Welcome Screen

With the Vivado Design Suite now open, you will load a helper Tcl script and run it to configure the project and get you to the important part of this lab—working with the MicroBlaze processor.

The Vivado Design Suite offers both GUI and scripted control. Scripted control takes the form of Tcl commands. These Tcl commands can be entered directly into the tool one at a time, or an entire Tcl script can be loaded and executed.

1-2. Run a Tcl script.

1-2-1. Locate the Tcl command line entry.

The command line entry can be found either on the Welcome page prior to a project being opened, or once a project has been opened.

From the Welcome screen:

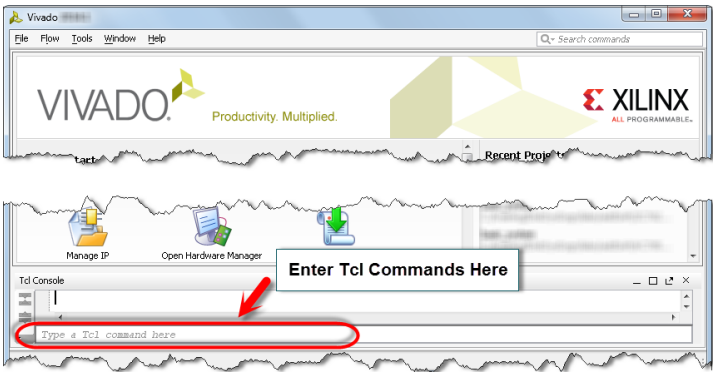


Figure 2-4: Accessing the Tcl Console from the Getting Started Page

From an opened project:

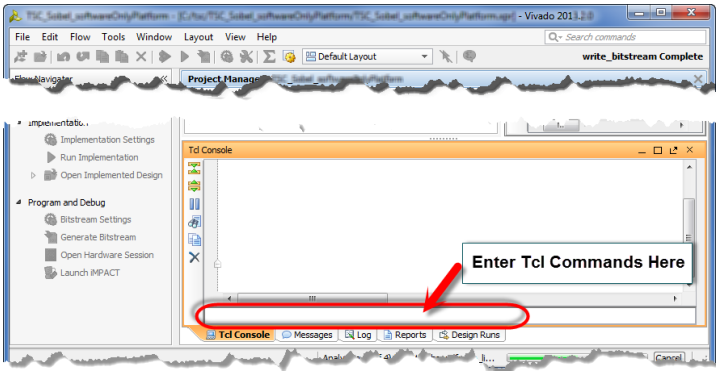


Figure 2-5: Entering Commands into the Tcl Console from an Open Project

The default directory for the Tcl environment is nested within the Xilinx installation directory. This placement, however, is often disadvantageous. In most cases, you will want to navigate to a more useful path. To do this, use the `cd` command to change directory to the user directory.

1-2-2. Change the current working directory to where the Tcl script is located by entering:

```
cd C:\training\ArchMicroBlaze_Overview\support
```

Remember that the Tcl environment is based on Linux and requires the '/' character to delimit hierarchical paths.

1-2-3. Verify that you are now where you want to be by entering the following into the Tcl command line:

```
pwd
```

The current working directory is displayed. If you are not where you want to be, use the `cd` command to change to `C:\training\ArchMicroBlaze_Overview\support`.

1-2-4. Enter the following Tcl command:

```
source exploreMB_completer.tcl
```

The Tcl script is run as though you typed each command included in the Tcl script into the Tcl command line. You can follow the execution of the script and monitor for any errors or warnings in the Tcl Console.

With the Tcl script now loaded, you will run the *createProject* and *createBlockDesign* procs to build the Vivado Design Suite project and create an appropriately named block design in which you will perform the remainder of this lab.

1-3. Run the *createProject* and *createBlockDesign* procs.

1-3-1. Enter the following into the Tcl command line to create the Vivado Design Suite project:

```
createProject
```

1-3-2. Enter the following into the Tcl command line to create the block design in which you will build your MicroBlaze processor system:

```
createBlockDesign
```

Adding and Customizing the MicroBlaze Processor Step 2

With the block diagram canvas open, you will now add and customize a MicroBlaze processor.

2-1. Add a MicroBlaze to the IP Integrator canvas.

If you do not recall how to perform this task, refer to the "Adding a Processor to the Block Design" section under IP Integrator Operations in the *Lab Reference Guide*.

Remember that adding a processor IP is just like adding any other type of IP to the canvas. You can also review the "Driving the IPI Tool" topic cluster for a guided example.

Next you will experiment with several different configurations to see how the MicroBlaze processor is realized in the FPGA/PL.

2-2. Open the MicroBlaze processor's Re-customization dialog box.

2-2-1. Double-click the **MicroBlaze** IP to open the Re-customization dialog box.

Alternatively, you can right-click the **MicroBlaze** IP and select **Customize Block**.

The MicroBlaze Configuration Wizard opens.

Question 1

What information does the Resources tab provide?

Question 2

Fill out the Default configuration in the table below.

As you investigate different configurations, you will add information to this table so that you can quickly see the tradeoffs between performance, area, and frequency.

Configuration Name	Frequency	Area	Performance
Default			
Minimum Area			
Maximum Performance			
Maximum Frequency			
Linux with MMU			
Low-End Linux with MMU			
Typical			

Configurations Table

Consider the following scenario: You are building a system on a small FPGA or Zynq® All Programmable SoC device. An accelerator (the most critical portion of this design) occupies a large portion of the total fabric/PL.

It is decided that you must add a MicroBlaze processor to this design for housekeeping tasks (such as user interface, communications, power monitoring, etc.) but not for computations.

Question 3

Given the above scenario, what configuration from the Predefined Configurations section would you choose and why?

2-3. Select the Minimum Area configuration.

2-3-1. Select **Minimum Area** from the Select Configuration drop-down list from the Predefined Configurations section.

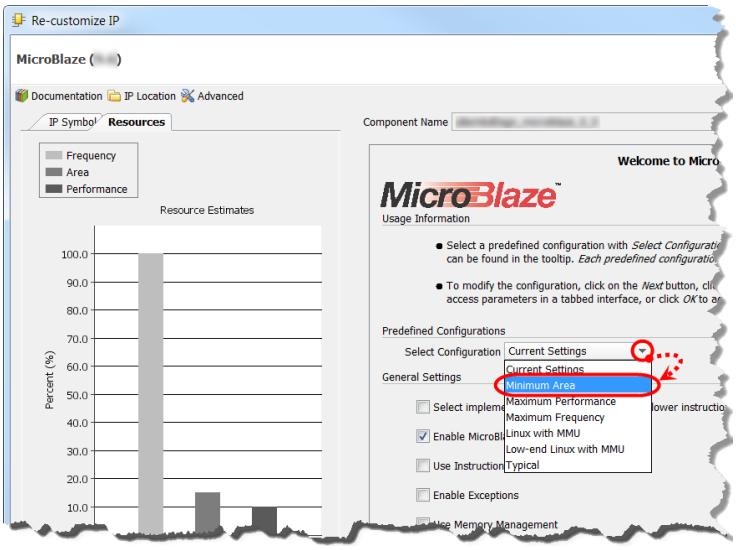


Figure 2-6: Selecting the Minimum Area Configuration

Question 4

Add the Frequency, Area, and Performance numbers to the Configurations table for the Minimum Area configuration.

As it turns out, there is a MicroBlaze processor in the accelerator portion of the design mentioned above. This particular MicroBlaze processor is responsible for preparing the data for the accelerator and throughput is paramount.

Question 5

For this scenario, what configuration might you select and why?

2-4. Select the Maximum Performance configuration.

2-4-1. Select **Maximum Performance** from the Select Configuration drop-down list from the Predefined Configurations section.

Question 6

Add the Frequency, Area, and Performance numbers to the Configurations table for the Maximum Performance configuration.

2-5. Select each of the remaining configurations and enter the values for the Frequency, Area, and Performance columns into the Configurations table.

2-5-1. Repeat the previous instruction but this time selecting the next pre-defined configuration from the Select Configuration drop-down list.

Question 7

Add the Frequency, Area, and Performance numbers to the Configurations table for each of the remaining configurations.

Question 8

Did you notice any changes to the General Settings section when the different configurations were selected?

Question 9

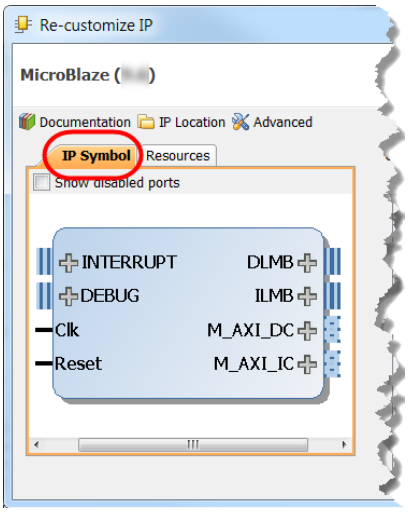
Where can you find an explanation for these options?

You should now have the Typical predefined configuration selected.

2-5-2. View the ports of the MicroBlaze processor to see what access points the MicroBlaze processor has to connect to the rest of the system.

2-6. Use the IP Symbol tab to view the graphical representation of the MicroBlaze processor in its current configuration.

2-6-1. Select the **IP Symbol** tab.



Figureb 2-7: Viewing the Graphical Representation of the Current Configuration of the MicroBlaze Processor

2-6-2. Click the '**+**' sign in the Interrupt interface to see the signals that comprise that interface.

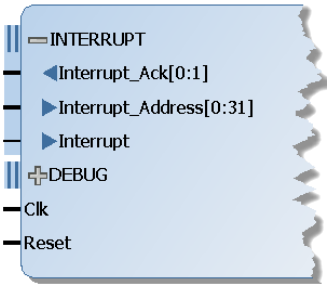


Figure 2-8: Expanded Interrupt Interface

This will show you the individual ports that comprise the Interrupt interface. You can explore other interfaces if you want.

2-6-3. Click the '**-**' sign next to the Interrupt interface to collapse the interface back to a single connection.

Question 10

When might the IP symbol in the IP Symbol tab change?

2-6-4. Click **Next** to advance to the next page of the re-customization.

2-6-5. Select the **Resources** tab to see how your choices affect the resources and performance estimations.

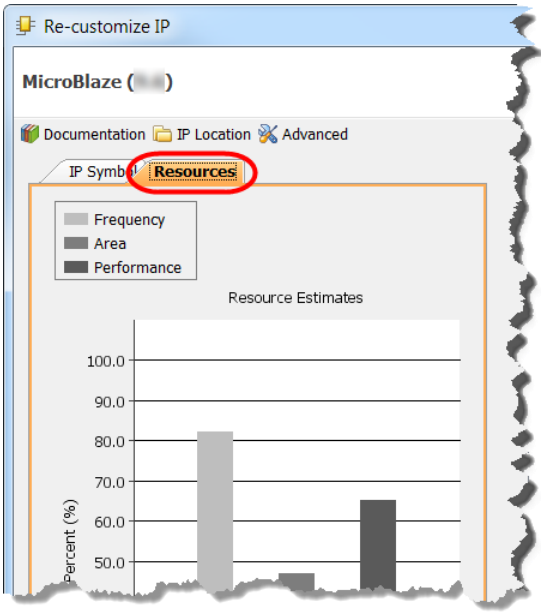


Figure 2-9: Selecting the Resources Tab During Re-customization of the MicroBlaze Processor

For the sake of this example, you can assume that the code will have sections requiring significant numeric processing of floating point numbers.

2-6-6. Select the **EXTENDED** mode for the FPU.

Question 11

How does one know when to use the different settings and options?

Question 12

How does selecting the FPU Extended mode change resource utilization and performance?

2-6-7. Click **Next** to advance to the page which customizes exception handling.

Since this design pre-supposes a lot of floating point activity, it makes sense to enable the FPU exceptions.

2-6-8. Select the **Enable Floating Point Unit Exceptions** option to enable these exceptions.

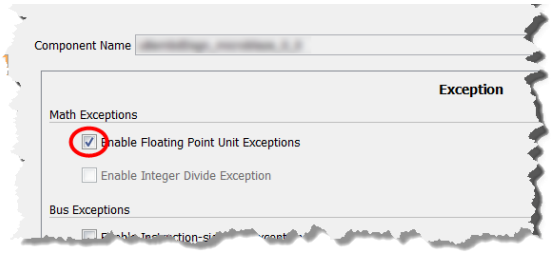


Figure 2-10: Enambling the FPU Exceptions

Question 13

How does enabling these exceptions affect the resource estimates?

2-6-9. Click **Next** to advance to the Cache configuration page.

Here you can enable or disable the instruction and/or data caches and control other parameters. This system will work acceptably without changing any of these parameters.

2-6-10. Click **Next** to advance to the Debug configuration page.

Because the MicroBlaze processor (and its associated support logic) is implemented in fabric/programmable logic, changes to the debug support can have significant effect on the resource estimates—particularly area.

2-6-11. Click **Next** to advance to the Buses page, the final page of the wizard.

Here you can enable the AXI and ACE interfaces, designate the number of streaming licks, and enable the trace bus interface (for debugging).

The local memory bus (LMB) interfaces are already enabled as part of the Typical configuration and are used to connect to block RAMs to provide the MicroBlaze processor with low-latency, user-manageable memory.

Note that the caches connect to the MicroBlaze processor via the M_AXI_DC and M_AXI_IC ports.

2-6-12. Click **OK** to accept the configuration and return to the IPI canvas.

Remember that you can re-configure the MicroBlaze processor at any time. However, use caution as you may need to re-run Designer Assistance to handle any changes you make.

Building a System around the MicroBlaze Processor Step 3

With the MicroBlaze processor now configured, it must be connected to supporting logic, including (but not limited to):

- Cache memory
- Clock
- Reset
- Interrupt controller
- Debug logic (MicroBlaze Debug Module - MDM)
- Local memory

You will now use Designer Assistance to run block automation.

3-1. Run block automation to build the supporting logic for the MicroBlaze processor.

While many pieces of IP will simply be connected when block automation is run, the MicroBlaze processor is a complex piece of IP and requires additional information so that block automation can assemble the properly configured system.

3-1-1. Click the **Run Block Automation** hyperlink in the information bar to launch the run bLock automation.

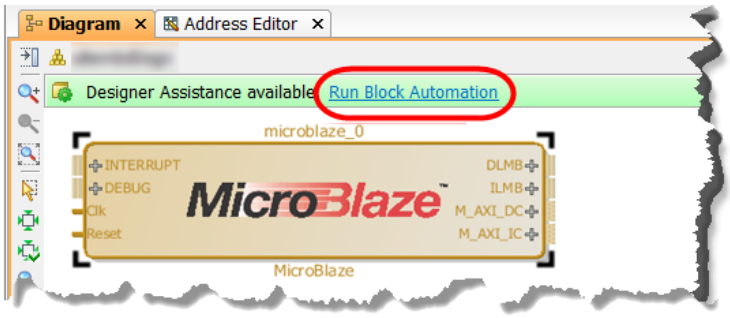


Figure 2-11: Launching the Run Block Automation for the MicroBlaze Processor

Note the current configuration as shown in the figure below.

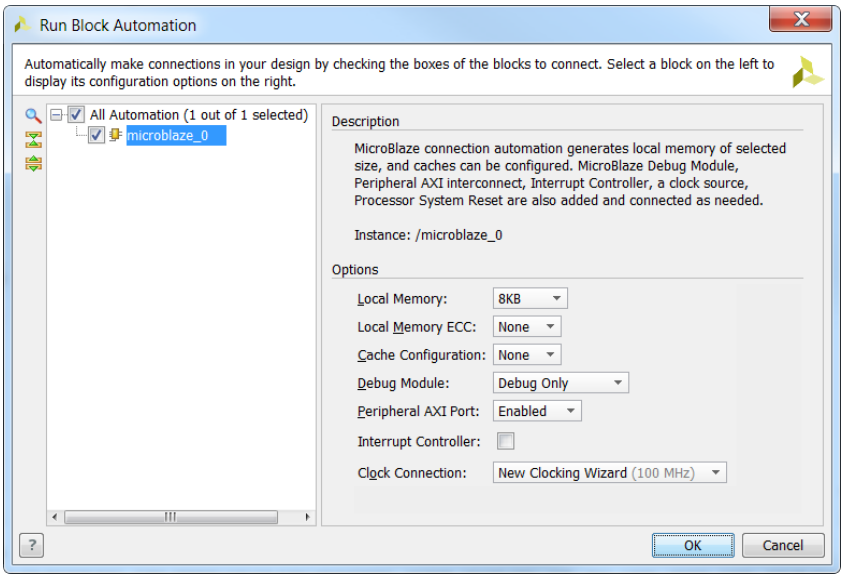


Figure 2-12: Default Settings for Run Block Automation for the MicroBlaze Processor

The MicroBlaze processor's Re-configuration Wizard does not communicate with the Run Block Automation tool.

3-1-2. Select **8KB** from the Cache Configuration drop-down list to change the cache configuration for each side.

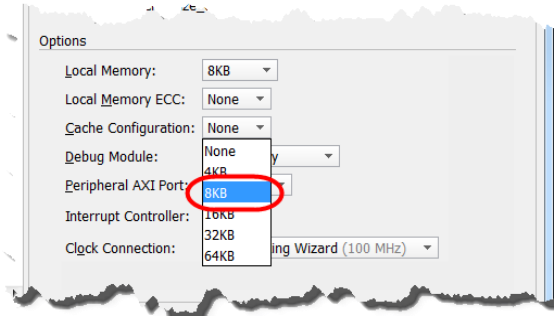


Figure 2-13: Selecting an 8KB Cache

3-1-3. Select the **Interrupt Controller** option to add an interrupt controller to the system.

3-1-4. Click **OK** to accept the options and run the Block Automation tool.

Question 14

What gets built?

While this system is perfectly adequate for responding to interrupts and performing basic processing, there a formal I/O channel does not exist.

3-2. Add the IP listed below to the block design.

If you do not recall how to perform this task, refer to the "Adding IP to an IP Integrator Block Design" section under IP Integrator Operations in the *Lab Reference Guide*.

- **AXI UARTlite for basic text I/O**

3-3. Customize the IP according to the table below.

If you do not recall how to perform this task, refer to the "Customizing IP" section under IP Integrator Operations in the *Lab Reference Guide*.

Baud Rate	19200
Data Bits	8
Parity	No Parity

You may note that there are no connections to the UART lite, nor to the Clocking Wizard, nor to the concat IP (which concatenates individual bits to form an input into the interrupt controller).

3-4. Run the Connection Automation tool to make the next set of connections.

3-4-1. Click the **Run Connection Automation** link in the status bar of the block design's Diagram tab.

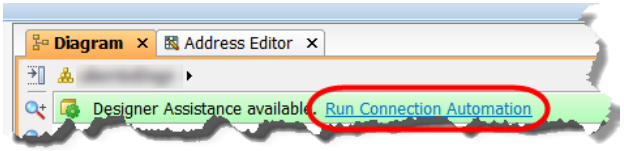


Figure 2-14: Launching Run Connection Automation

The Run Connection Automation dialog box opens and invites you to identify the IP that you want to run the connection automation for.

3-4-2. Select the **All Automation** option.

This will cause all the options to be selected.

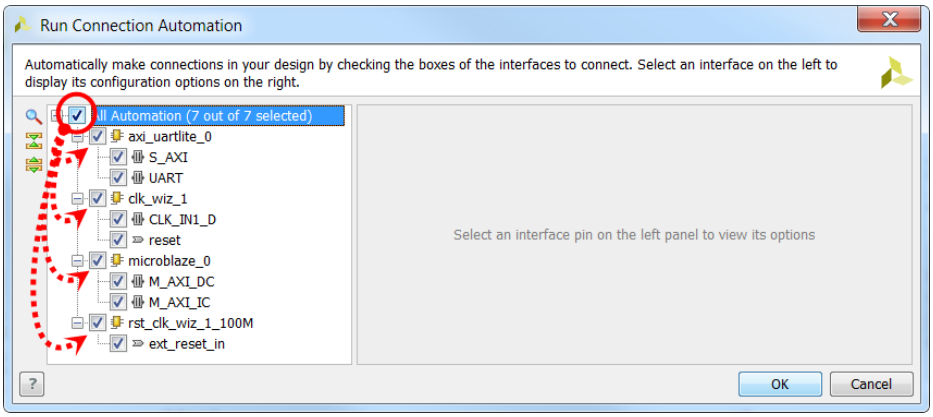


Figure 2-15: Selecting the All Automation Option

3-4-3. Click **OK** to run the Connection Automation tool.

3-4-4. [Optional] Click the **Regenerate Layout** icon ( ) in the left-hand vertical toolbar to redraw the current system.

Generally, this will cause the system to be redrawn with inputs on the left-hand side of the canvas and outputs on the right-hand side.

Question 15

Are there any unconnected inputs and/or outputs?

3-5. Remove the microblaze_o_xlconcat IP from the design and connect the interrupt output of the AXI Uartlite into the AXI Interrupt Controller's input.

The purpose of this IP is to collect multiple interrupts and merge them into a single bus for the AXI Interrupt Controller. Since there is only one interrupt signal (from the UART), this IP is no longer needed.

3-5-1. Select the **microblaze_0_xlconcat** IP.

3-5-2. Press <**Del**> to delete the selected item.

Note that this also removes any nets connected to this IP.

3-5-3. Click-drag a connection from the axi_uartlist_0 interrupt port to the microblaze_0_axi_intc.intr[0:0] port.

3-6. Validate the design to catch any connection and address map errors.

3-6-1. Select **Tools** > **Validate Design**.

Any issues are displayed in the Console window.

3-6-2. Look for any errors—there should be none.

3-6-3. Click **OK**.

Question 16

Is this now a complete system?

Summary

You just completed developing a MicroBlaze processor-based embedded design. You explored the MicroBlaze processor's Re-customization Wizard and the Run Block Automation options for rapidly constructing a viable system. A UART was also added to this system to extend its capabilities. You could have continued to add peripherals and memories in the same basic fashion that the UART was added.

Answers

1. What information does the Resources tab provide?

   The Resources Tab shows a bar chart indicating the relative performance of the MicroBlaze processor as configured by the current settings in the configuration wizard compared to the maximum frequency that the device can run compared to the amount of area that the processor will occupy in the FPGA/PL.

   The default settings show that this configuration will run at the fastest clock rate (f_{max}) possible for any configuration of the MicroBlaze processor. Additionally the settings show that the performance is about 1/10th of the highest performing configuration of the MicroBlaze processor. Finally, the settings show that it will take up about 12% of the area of the largest MicroBlaze processor configuration.

2. Fill out the Default configuration in the table below.

Configuration Name	Frequency	Area	Performance
Default	100%	13%	10%

3. Given the above scenario, what configuration from the Predefined Configurations section would you choose and why??

   The best choice would be the Minimum Area configuration so that it has the best chance of fitting into the remaining logic. Since this MicroBlaze processor is only performing housekeeping chores, performance and frequency are secondary concerns to that of area.

   Even if the rest of the logic is running at a higher clock speed, this MicroBlaze processor can take its clock from one of the many clock dividers (PLLs) on the device and run at a lower speed than the other IP on the device.

4. Add the Frequency, Area, and Performance numbers to the Configurations table for the Minimum Area configuration.

Configuration Name	Frequency	Area	Performance
Default	100%	13%	10%
Minimum Area	87%	9%	10%

5. For this scenario, what configuration might you select and why?

   Since throughput (performance) is the most important factor, the Maximum Performance configuration will likely be the best choice.

6. Add the Frequency, Area, and Performance numbers to the Configurations table for the Maximum Performance configuration.

Configuration Name	Frequency	Area	Performance
Default	100%	13%	10%
Minimum Area	87%	9%	10%
Maximum Performance	61%	61%	100%

7. Add the Frequency, Area, and Performance numbers to the Configurations table for each of the remaining configurations.

Configuration Name	Frequency	Area	Performance
Default	100%	13%	10%
Minimum Area	87%	9%	10%
Maximum Performance	61%	61%	100%
Maximum Frequency	100%	13%	10%
Linux with MMU	58%	57%	71%
Low-End Linux with MMU	61%	47%	57%
Typical	68%	32%	61%

8. Did you notice any changes to the General Settings section when the different configurations were selected?

   The options in the General Settings section enable you to modify the predefined configurations to better suit your needs.

9. Where can you find an explanation for these options?

   Vivado Design Suite User Guide: Embedded Processor Hardware Design (UG898), Chapter 4.

   Appendix A (Performance and Resource Utilization) of the *MicroBlaze Processor Reference Guide* (UG984) also contains actual frequencies and LUT utilizations for the different families.

10. When might the IP symbol in the IP Symbol tab change?

    Any configuration that changes the number or types of ports will cause the IP symbol to be updated to show the new ports. The General Settings > Enable Discrete Ports option makes a number of the control and sense ports available.

    Similarly, the Show Disabled Ports option shows all of the possible ports on the MicroBlaze processor (a superset of all configurations). The active ports are listed in black and the disabled ports are shown in gray.

11. How does one know when to use the different settings and options?

    It is generally a good rule to use one of the pre-defined configurations as a starting point. Then based on the system constraints (such as limited resources or clocking speeds) or code constraints (heavy use of floating point numbers, integer multiplication and division, etc.) the MicroBlaze processor can be tailored to meet specific goals.

    Before experimenting with these options, be sure to read the *MicroBlaze Processor Reference Guide* (UG984).

12. How does selecting the FPU Extended mode change resource utilization and performance?

    Prior to the enabling of the FPU, the frequency was about 81%, area about 32%, and performance was at about 61%.

    Afterwards, the frequency remains the same at about 81%; however, the area increases to about 48% and the performance increases to about 65%.

    An interesting point to think about: If your code does not perform any floating point computations, your system's overall performance will not change. However, if your system is performing a lot of floating point computations, then the system's actual performance may be significantly higher. Refer to the "Amdahl's Law" topic cluster to better understand how accelerating one part of the code affects the entire system's performance.

13. How does enabling these exceptions affect the resource estimates?

    Frequency is reduced from about 81% to about 65%. Neither area nor performance changes significantly.

14. What gets built?

    The Run Block Automation tool adds and connects the following IP to the system:

    - Cocking Wizard: Provides a 100-MHz clock to the MicroBlaze processor system.
    - Processor System Reset: Manages both internal and external signals to reset the system. It produces five resets (of which one is not used): One for the peripherals, one for the interconnect controller, one for the bus structure (and is used by the local memory), and one for the processor.
    - MicroBlaze Debug Module: Connects to the chip's BSCAN (JTAG) chain and provides debugging support for the MicroBlaze processor.
    - Interrupt Controller: Manages interrupts from various hardware sources and produces an interrupt signal to the MicroBlaze processor.
    - AXI Interconnect: An AXI *switch* that enables various AXI masters (in this case, the MicroBlaze processor) to connect to a number of slave device or peripherals (in this case, the Interrupt Controller IP).
    - Local Memory: A collection of block RAMs that house local instruction and data. Small applications can be run completely from this local memory.

15. Are there any unconnected inputs and/or outputs?

    Yes. the concat IP (which collects interrupts and provides them not as single signals, but as a bus to the Interrupt Controller IP) wants two interrupts and the AXI Uartlite has an unconnected output.

16. Is this now a complete system?

    Yes, with one significant consideration. The MicroBlaze processor was configured to use caches. These AXI cache lines are run into the AXI Interconnect; however, there are no memory devices attached to the AXI Interconnect, which renders the cache connections useless!

    You can optionally add block RAMs to the AXI Interconnect for the caches. Even so, caching is beneficial only when there is a slower memory, such as DDR. You are better off removing the caching capability of the MicroBlaze processor and enlarging the block RAMs attached to the local memory bus (LMB).

    If there is a shortage of block RAM (that is, not enough block RAM to hold data and instructions), you would need to instantiate a MIG (Memory Interface Generator) to connect to off-chip DDR memory and instantiate block RAMs for cache.

    Even as is, this design could work. Just as any FPGA/SoC/MPSoC design requires when there are signals entering or leaving the logic/PL, constraints are required to properly map clock, reset, and communication signals in and out of the device.

Lab 3: Driving the SDK Tool

2016.3

Abstract

This lab introduces you to the basic operations for SDK. Concepts such as project creation, adding existing source code to an application, compilation and linking, and downloading are covered. Numerous other labs focus on other aspects of the SDK tools.

Objectives

After completing this lab, you will be able to:

- Create an SDK workspace
- Create the three basic types of projects typically required for application development
- Navigate application source code
- Produce and download an ELF to a hardware platform

Introduction

SDK is a powerful and complex tool. Knowing how to navigate its basic capabilities will make software development a much easier task.

First, you must understand a few basic SDK concepts, starting with the workspace. The workspace is simply a location (usually a directory) that contains not only all of the projects (each in its own subdirectory), but also a few SDK files that help SDK remember user settings and how projects relate to each other.

Next is the concept of the project. A project is a collection of related files for a certain type of thing. Some projects that SDK intrinsically knows about and that you will use in this lab: the hardware platform specification, board support package, and application.

The following figure illustrates the relationships among these three primary projects:

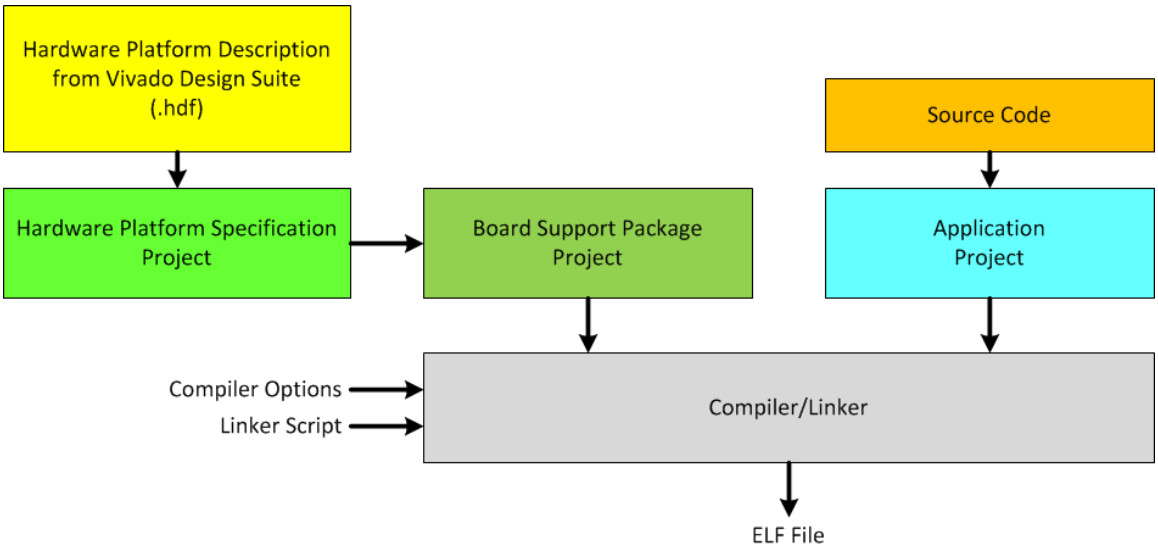


Figure 3-1: Relationships Among the Three Primary Project Types

You will begin by creating a new workspace and, within that workspace, one of each of the commonly used projects.

As the diagram suggests, you will begin by importing a hardware description file (HDF) which was created with the Vivado® Design Suite that will be used to create the hardware platform specification project. Next you will create an application project (and, as part of the process, you will generate a board support package based on the hardware platform specification project).

With the three projects created, you will add provided source code to the application project and compile, link, and load it to generate an ELF (executable load format) file. Since the provided source code contains some errors, you will correct them and recompile. With the errors corrected, the resulting ELF file can then be downloaded onto the board and run.

General Flow

Launching SDK and Creating the Hardware Specification Project Step 1

You will begin this lab by open the SDK tool, identifying its workspace (where the projects will be kept on disk), and creating a hardware specification project that describes the hardware system that the software will be running on from a file provided to you which was exported from the Vivado Design Suite.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\SDK_Driver\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\SDK_Driver\lab** directory and click **OK**.

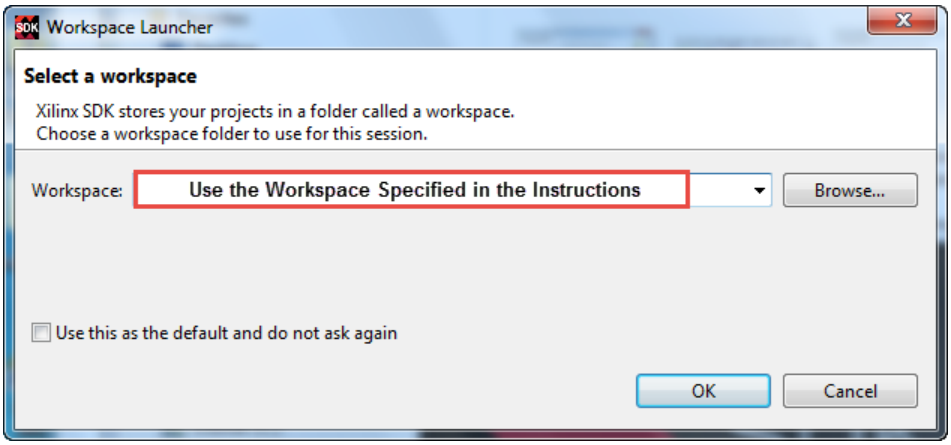


Figure 3-2: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

The hardware platform specification contains a thorough description of the hardware design: what types of processors are present, active peripherals in the PS and PL for Zynq® All Programmable SoC-based systems or a list of all peripherals for a non-Zynq All Programmable SoC system, a full system memory map, etc. Based on this description, software such as the board support package (BSP) and application can be tailored to the hardware.

1-2. Create the hardware platform specification.

1-2-1. Select **File** (1) > **New** (2) > **Other** (3) to see the Xilinx-specific options.

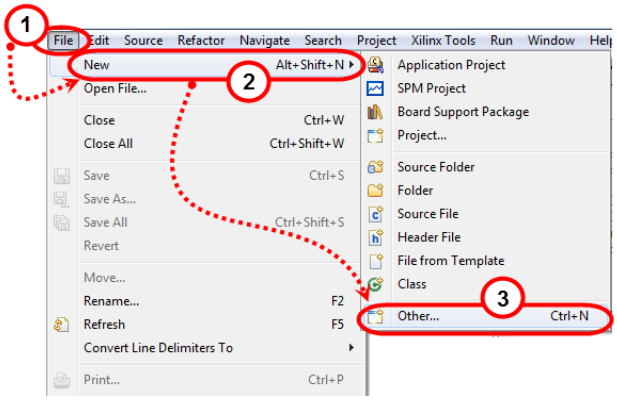


Figure 3-3: Accessing the New Wizards

The Select a Wizard dialog box opens. Here you can select one of many different wizards.

1-2-2. Expand the **Xilinx** folder (1).

1-2-3. Select **Hardware Platform Specification** (2).

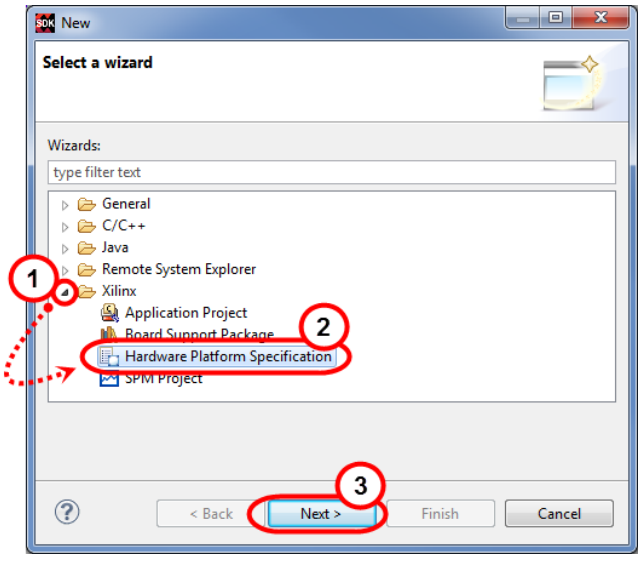


Figure 3-4: Selecting the Hardware Platform Specification Wizard

1-2-4. Click **Next** to open the New Hardware Project dialog box (3).

The New Hardware Project dialog box opens. Here you will be able to specify a project name and the hardware description file that was exported by the Vivado Design Suite.

1-2-5. Enter **SDKintro_hw** in the *Project name* field.

1-2-6. Browse to the *C:\training\SDK_Driver\support* directory under the Target Hardware Specification region and select the **blkdsgn_[ZC702 | Zed]_[A9 | MicroBlaze].hdf** file.

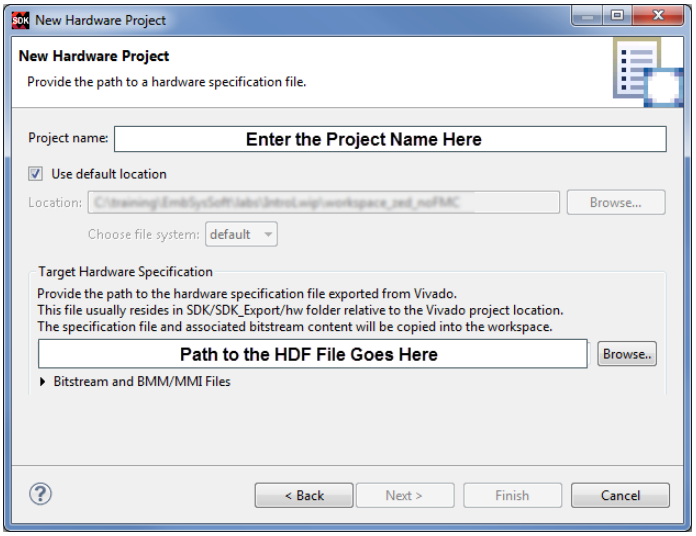


Figure 3-5: New Hardware Project Dialog Box

1-2-7. Click **Finish** to create the new hardware project.

Creating the Application Project and Navigating Code Step 2

With the SDK tool open and the hardware description imported, you will now create a new application and, as part of creating the new application, you will create a board support package with the default settings.

While there are a number of templates available during the creation of an application project, you will use the Empty Project template and import the provided source code files.

Using the Application Project Wizard is a quick way to set up a C or C++ software application project that targets an existing processor and OS platform (Standalone or Linux). You can automatically generate the board support package (BSP) or select an existing one. Based on the dialog box choices, the appropriate tool chain is selected for pre-processing, compiling, assembling, and linking.

2-1. Create a new C/C++ application project named *SDKintro_app*. Use the board support package named *default_bsp*.

2-1-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

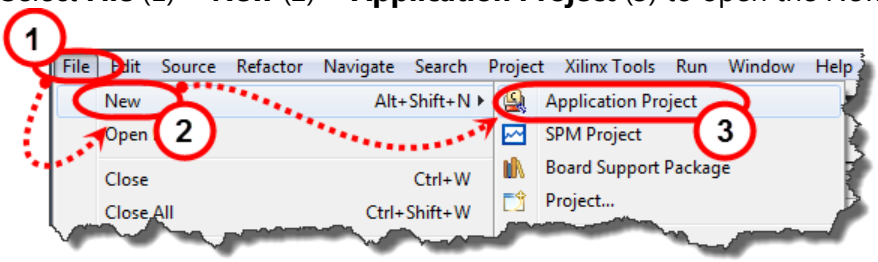


Figure 3-6: Creating an Application Project

2-1-2. Enter **SDKintro_app** as the project name.

2-1-3. Ensure that you have **SDKintro_hw** selected from the Hardware Platform drop-down list as the SDK tool can manage multiple platforms within a single workspace.

This will populate the Processor drop-down list accordingly.

2-1-4. Ensure that **ps7_cortexa9_0 or microblaze_0** is selected from the Processor drop-down list.

2-1-5. Ensure that **standalone** is selected from the OS Platform drop-down list.

2-1-6. Select **Use Existing** and choose an existing BSP from the drop-down list if you have already created a BSP for this hardware platform; otherwise, select **Create New** and enter the name for the BSP as **default_bsp**.

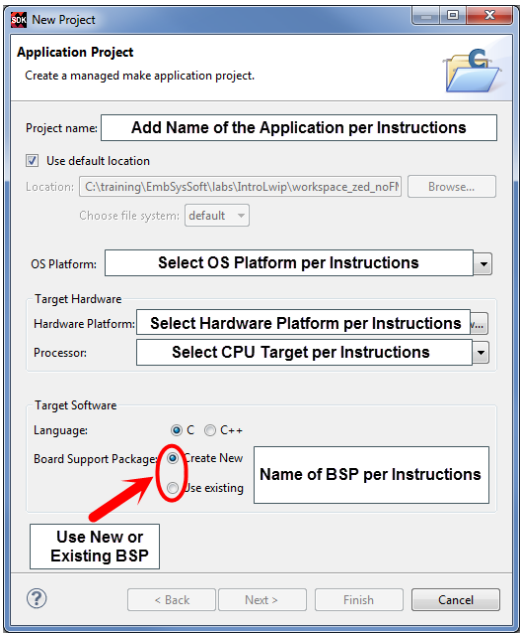
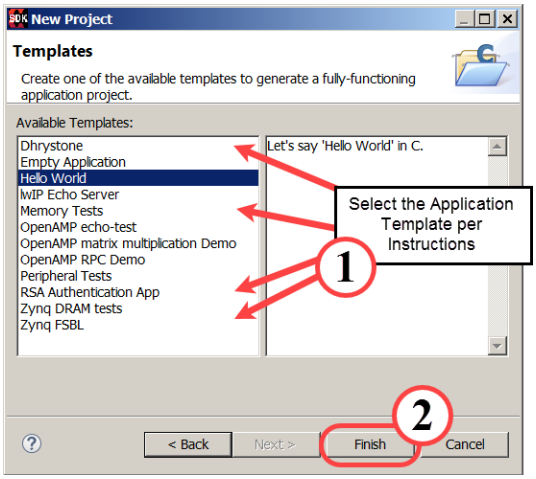


Figure 3-7: Entering Application Project Information

2-1-7. Click **Next** to select the template for this application.

2-1-8. Select **Empty Application** (1).



Figureb 3-8: Selecting an Application Template (Selection in Figure May Not Match Instructions)

2-1-9. Click **Finish** to create the new project (2).

As the project is created, the new BSP is compiled and any sources from the templates are also compiled. The creation of the new project usually takes less than a minute.

The application framework is generated and now it is time to load the provided source files.

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

2-2. Add *SDKintro_main.c, hardware_support.c, hardware_support.h, LED_drivers.c, LED_drivers.h, platform.c, platform.h, platform_config.h, serialPort_helper.c, serialPort_helper.h, utils_print.c, and utils_print.h* to the application.

The preferred method for importing sources is shown here.

2-2-1. Expand the project named **SDKintro_app** > **src** using the Project Explorer.

2-2-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

2-2-3. Select **Import** to open the Import Wizard (2).

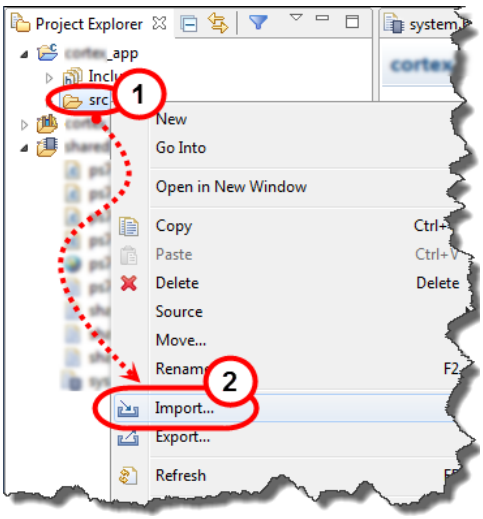


Figure 3-9: Importing a Resource File

The Import Wizard dialog box opens.

2-2-4. Expand **General** (1).

2-2-5. Select **File System** as you will be selecting individual files directly from the file system (2).

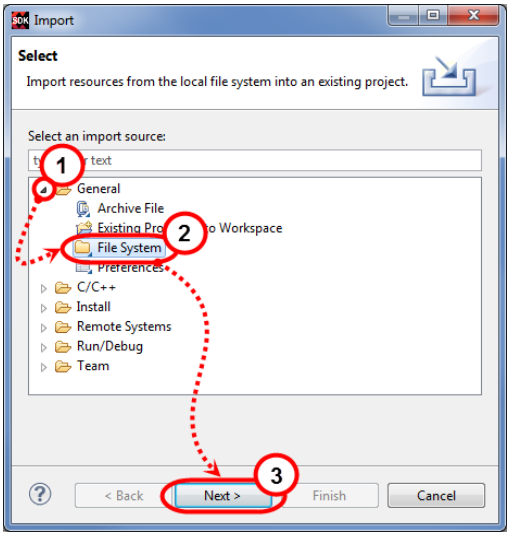


Figure 3-10: Selecting File System

2-2-6. Click **Next** to advance to specifying the files to import (3).

2-2-7. Browse to *C:\training\SDK_Driver\support* in the *From directory* field.

2-2-8. Select the file(s) by checking the box beside **SDKintro_main.c, hardware_support.c, hardware_support.h, LED_drivers.c, LED_drivers.h, platform.c, platform.h, platform_config.h, serialPort_helper.c, serialPort_helper.h, utils_print.c,** and **utils_print.h**.

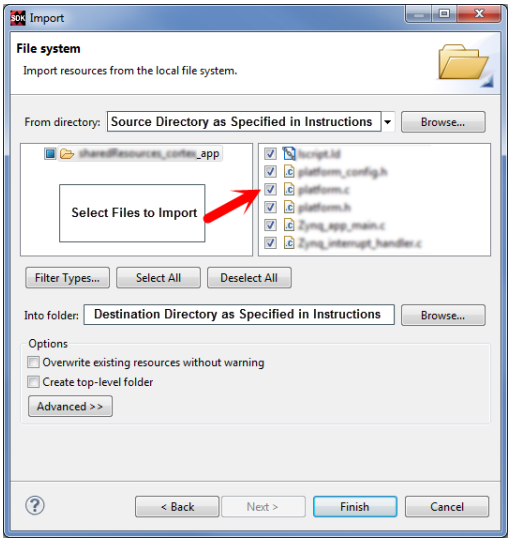


Figure 3-11: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

2-2-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

The Project Explorer tab shows a tree structure of the various projects in this workspace. Each entity is its own project; that is, the hardware platform specification is a complete project and contains the XML description of the hardware and other necessary hardware files (usually including a bitstream).

The BSP is another project that is dependent on the hardware platform specification and contains various libraries, including device drivers for the standalone configuration. The BSP project is dependent upon a hardware platform project.

Finally, the application becomes yet another project and contains the source files, linker script, compiler options, etc. necessary for running the software. The application project is dependent upon a BSP project.

Multiple projects of any type can exist in the workspace. For this lab, *SDKintro_app* is the application, the *SDKintro_app_bsp* project is the BSP that contains all of the supporting software that bridges the hardware to the application software, and the *SDKintro_hw* project is the description of the hardware platform.

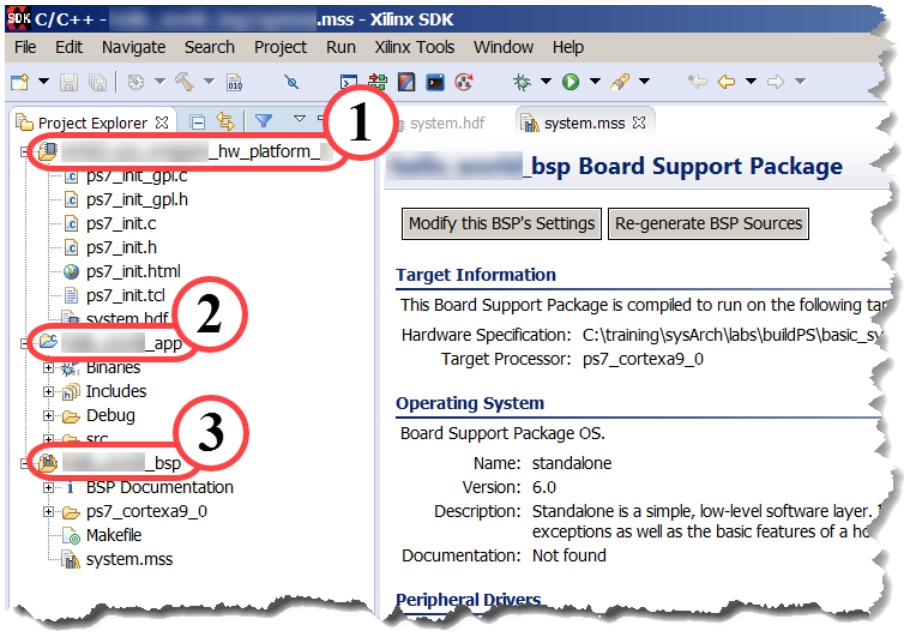


Figure 3-12: Project Explorer Tab

2-2-10. Expand **SDKintro_app** > **src** to see all of the source files that are part of this project by clicking the ▷ or ⊞ sign next to src.

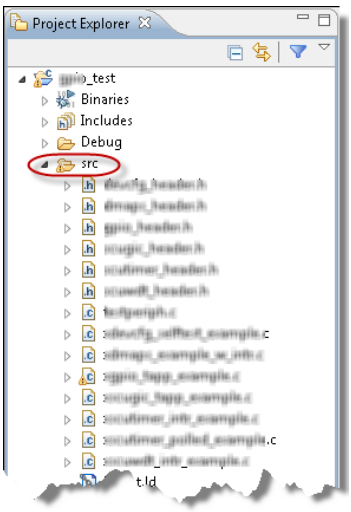


Figure 3-13: Expanding the Application Sources Structure

With the new sources added to the project, the SDK tool will automatically compile all the files in the project. Since errors was deliberately introduced into the source code, your next task is to locate and correct these errors.

2-3. Locate the first error and correct it.

There are several ways to locate errors. You will use one method to locate and correct this first error and another process to locate and correct the second error.

2-3-1. Locate the **Problems** tab.

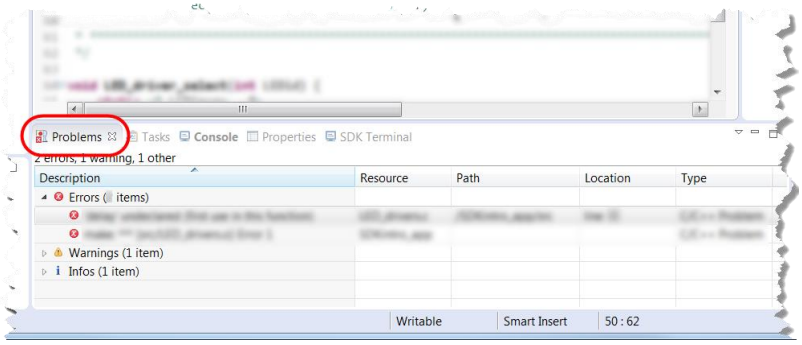


Figure 3-14: Locating the Problems Tab

This tab is found at the bottom of the workspace.

2-3-2. Double-click the first error ***'delay' undeclared (first use in this function)*** to both open the file that contains this error and identify the line where the error occurred.

Notice the two small icons in the margin (⊗ and ⚠). These indicate errors and warnings.

2-3-3. Hover the mouse over each of the icons to activate the pop-up message describing what the warning and error are.

Figure 3-15: Accessing the Pop-Up Description for the Warning and Error

The error icon shows that something in this line is not defined. The description in the Problems tab clarifies this and tells you that 'delay' has not been declared.

Question 1

What action might you take to resolve this issue?

Certain types of problems can *propagate* from an error or warning to others.

Question 2

What is the best way to investigate and correct this error?

Question 3

Unfortunately, the default for the SDK editor is to have line numbering turned off. How can you turn on line numbering?

2-3-4. Change **dely** to **delay** on the line prior to the error (on or about line 49).

2-3-5. Select **File** > **Save** (or press <**Ctrl** + **S**>) to save your changes.

Saving the file also causes the rebuilding of the file. This is a customizable feature and can be disabled.

Question 4

What happened as a result of the recompilation of the file? **Hint:** Look at the Problems tab.

2-4. Locate and correct the second error.

While you could follow the same procedure described in the previous instruction, you will use this second error to explore an alternate means of performing the same task. One method is just as good as the other—this will simply become your preference.

2-4-1. Expand **SDKintro_app** > **src** under the Project Explorer tab.

2-4-2. Locate the file icon with an error marker ( ).

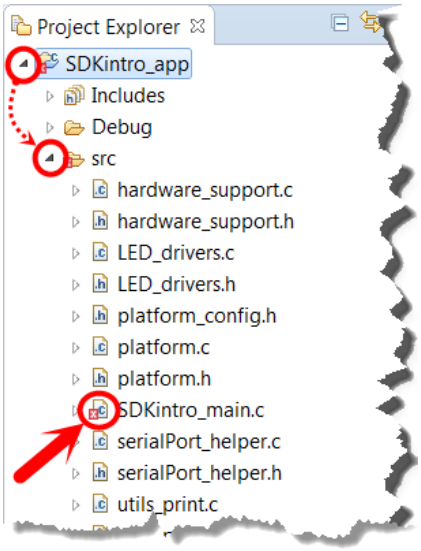


Figure 3-16: Locating a Source File Containing an Error

2-4-3. Double-click the file name to open it in the editor.

2-4-4. Look closely at the right margin to notice a series of small red and yellow rectangles.

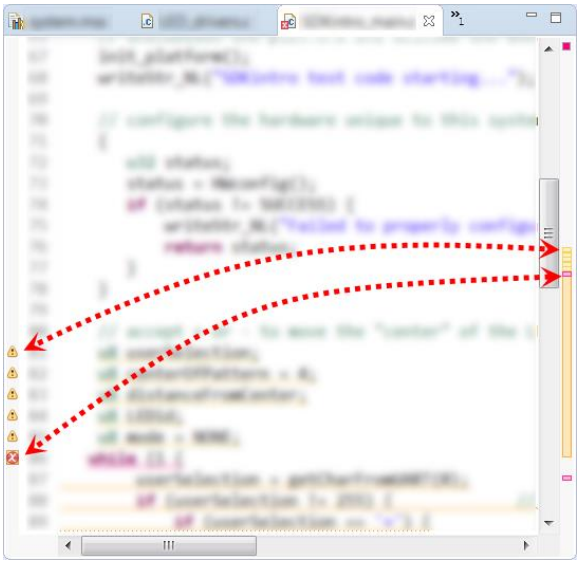


Figure 3-17: Mapping Error and Warning Icons to the Error and Warning Navigation Bar

Question 5

What do you think these small red and yellow rectangles mean?

2-4-5. Click the topmost red rectangle in the right margin to change the view to show the offending line of code.

Visual inspection quickly shows that the closing parenthesis was omitted from the *while* statement.

2-4-6. Insert the closing parenthesis on (or about) line 86 so that it appears as follows:

```
while (1) {
```

Question 6

What happens as soon as you add the closing parenthesis?

2-4-7. Select **File** > **Save** (or press <**Ctrl** + **S**>) to save the code to preserve your changes and recompile the source code.

A successful build will show no errors in the Problems tab (although there might be warnings) and is indicated in the Console tab when the sizes of all of the program segments are displayed before the *Build Finished* message.

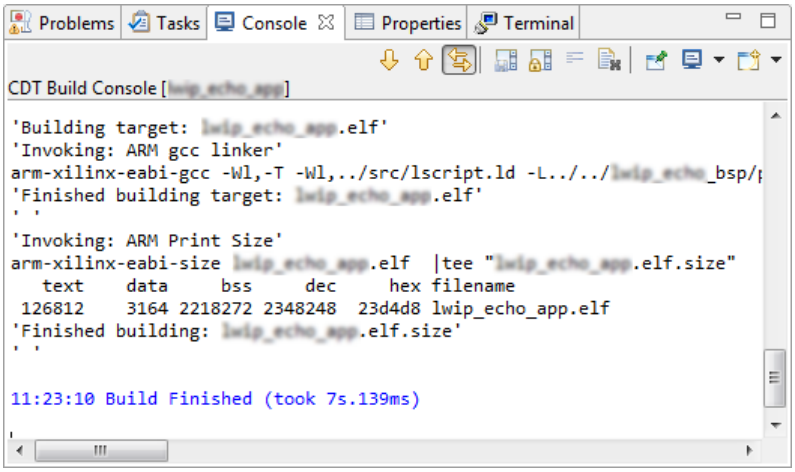


Figure 3-18: Successful Software Application Build (Your File Names and Sizes Will Vary)

Sometimes you want to confirm that everything has built properly and you want to "force" everything to be rebuilt.

2-5. Force a recompilation of your entire project.

2-5-1. Select **Project** > **Build All** to rebuild the entire project.

You will be able to monitor the progress in the Console tab and in a pop-up status bar.

The Outline tab (view) is the result of the tools scanning the source code and mapping where a number of different elements are located. These elements include:

- *#define*
- *#include*
- Global variable declarations
- Function definitions

This list is filterable so that searching can be simplified for large and complex sources.

2-6. Use the Outline view to answer the following questions.

Question 7

How do you use the Outline view?

Question 8

What line number does the *LED_driver_select(int)* function begin on? (Located in LED_drivers.c)

Question 9

What happens if you click a different source tab?

2-6-1. Select the **SDKintro_main.c** source to to make it active.

2-6-2. Using the Outline view, locate the include for *utils_print.h*.

2-6-3. Select **utils_print.h** in the Outline view to locate the *include* in the source file.

Question 10

What is the easiest way to open *utils_print.h*?

Question 11

What else turns into a hyperlink?

Configuring the Serial Port and Running the Application Step 3

You will now take the final step and download the ELF file to the hardware and monitor the output of the system.

Begin by powering up the board and initiating a serial communications session.

3-1. Apply power to the board.

3-1-1. Switch the board to OFF to clear memory if your board is powered on and has a power slide switch; otherwise, remove the power cable.

3-1-2. Ensure that you have a correctly imaged SD card if you are using the MicroBlaze™ processor.

The correct SD card images are available from *C:\training\SDK_Driver\support\TransUartPins48-49_BootImage\[ZC702 | ZED]*. Copy the BOOT.bin to the root of the SD card.

3-1-3. **MicroBlaze processor users only:** Insert the SD card into the development board.

This will configure the PS with a special application to allow UART communications with the MicroBlaze processor through the PS.

Note: You may have to change the jumper/switch settings to support JTAG programming. Many devices, such as the Zynq®-7000 family support JTAG fallback if the jumpers/switches are set to SD card and no card is present. Other families, such as the Zynq UltraScale+™ family, need to be specifically configured to boot from JTAG or SD Card. Refer to the lab setup guide for your board for details.

3-1-4. Slide the switch on your board to the ON position if it has one (as shown in the lab setup guide for your board); otherwise, plug the power cable in.

You will need to verify the proper connections and open a terminal window on the PC to monitor the output of the application. You may need to change the assigned COM port based on your computer's configuration.

3-2. If you are not certain on how to properly connect your board to the PC and power it up, refer to the Board-Specific Tasks section in the *Lab Reference Guide*.

If you do not recall how to determine the COM port that is connected to the board, refer to the "Determining the COM Port Assigned by the USB Driver" topic under the Board, OS, COM, and IP Address Tasks section in the *Lab Reference Guide*.

If you need further assistance with either of these tasks, ask your instructor.

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

3-3. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

3-3-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

3-4. Configure the SDK Terminal.

3-4-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

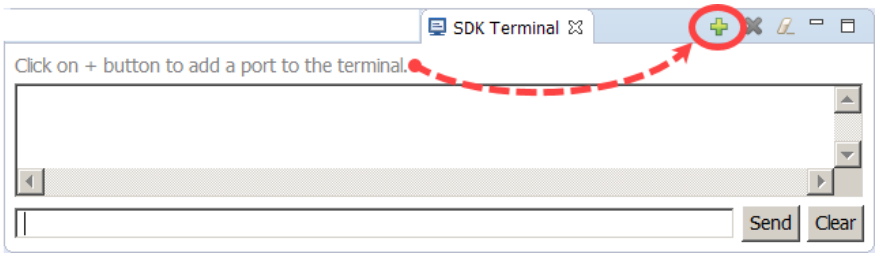


Figure 3-19: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

3-4-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

3-4-3. Set the baud rate to **115200** (2).

3-4-4. Leave the other settings at their defaults.

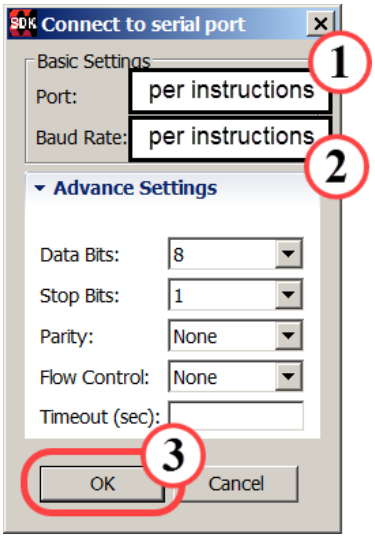


Figure 3-20: Configuring the SDK Terminal

3-4-5. Click **OK** to save these settings and begin the terminal session (3).

Note: The hardware design used for this lab contains a Programming Logic (PL) component for both the MicroBlaze design and the Zynq based design. For the Zynq design this allows access to the on-board LEDs through a GPIO IP Block in the PL. For the MicroBlaze™ design, this contains the softcore MicroBlaze™ processor, the Uart IP Block and the GPIO IP Block for LED access.

3-5. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

3-5-1. Select **Xilinx Tools** > **Program FPGA** or click the icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM

format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

3-5-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

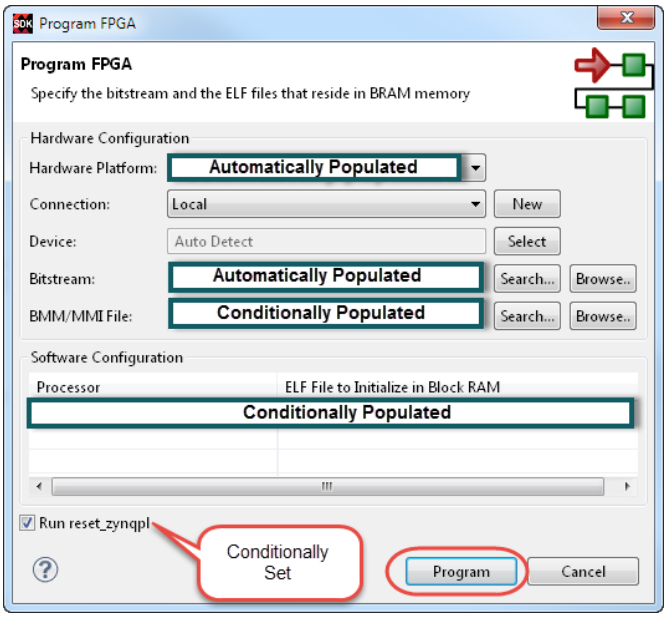


Figure 3-21: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

Note: If you are programming a MicroBlaze softcore processor on a Zed or ZC702 Development Board a warning indicating that there is not a PS in the design will be presented. Read the warning and click OK to dismiss it. The design does not use any PS managed or initialized resources.

Now that you are ready to receive application output, it is time to launch the application by creating a new run configuration.

A Run configuration defines how you want the system to work when running an application. While there are a significant number of switches and options, the most common are shown below.

3-6. Set up a Run configuration for *SDKintro_app*.

3-6-1. Right-click **SDKintro_app** (1) from the Project Explorer pane.

3-6-2. Select **Run As** from the context menu (2).

3-6-3. Select **Run Configurations** (3).

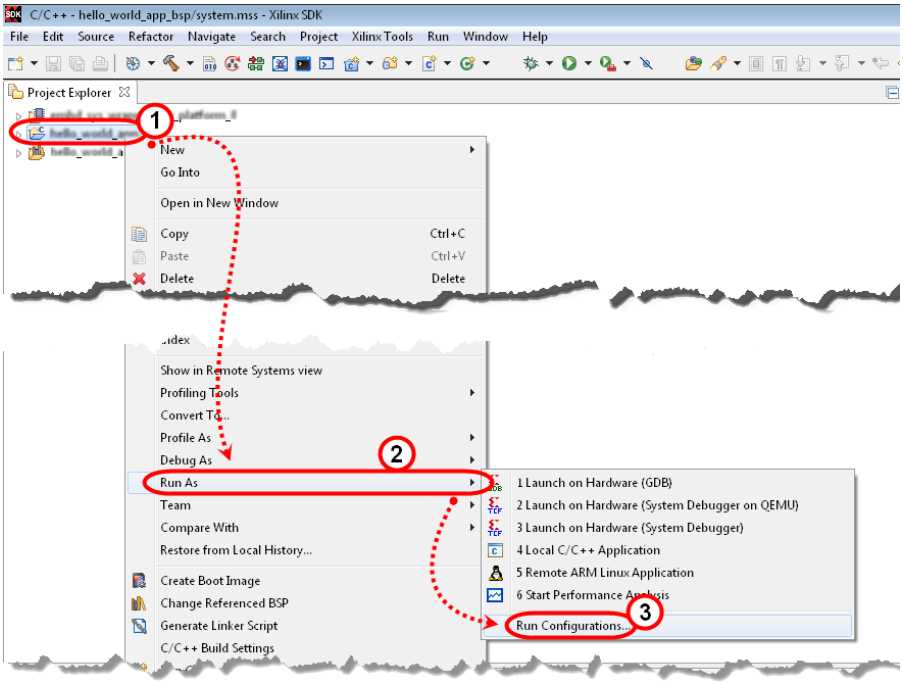


Figure 3-22: Creating a Run Configuration for an Application

The Run Configurations window opens.

3-6-4. Double-click **Xilinx C/C++ application (System Debugger)** to create a launch configuration (1).

Alternatively, you can select **Xilinx C/C++ application (System Debugger)** and click the **New** configuration button.

If this is the first debug configuration being created for the project, a welcome type dialog opens. If one or more configurations exist, then the last open configuration will be displayed. In either case, a new configuration can always be added. Existing configurations are shown in the left pane and can be selected for debugging.

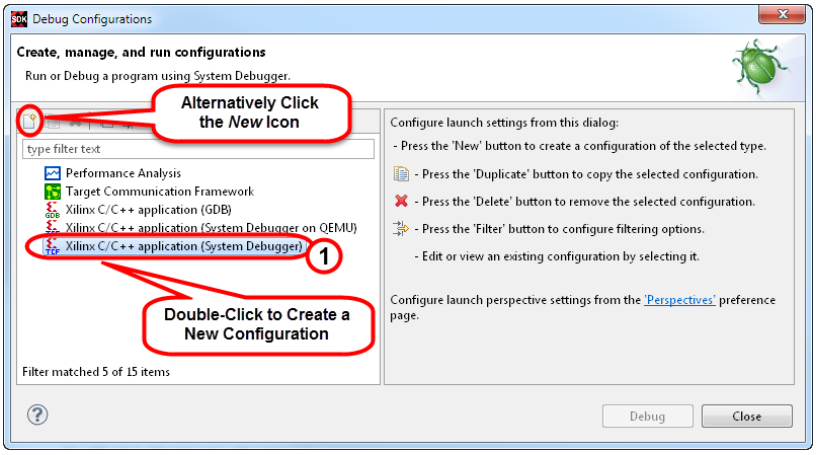


Figure 3-23: Creating a New System Run/Debug Configuration

A new Run Configuration is created named *System Debugger using Debug_SDKintro_app.elf on Local* (2).

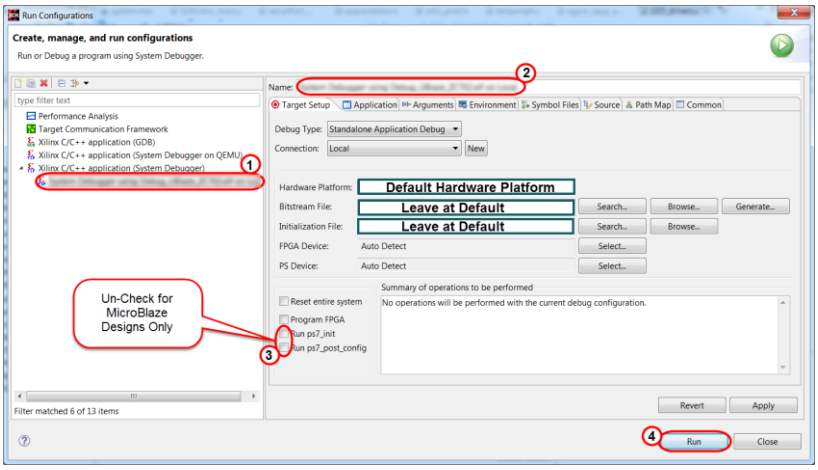


Figure 3-24: Run Configurations Dialog Box (Zynq AP SoC)

The new configuration will appear with other existing configurations and have the name of your application. You will also note that a number of fields are automatically filled in for you using the name of your application as the basis (that is, if your application is named "XYZ", then the Name field will be populated with XYZ Debug.) Most users will leave the other settings at their default.

3-6-5. Deselect the **Run ps7_init** and **Run ps7_post_config** options (3) if you are targeting a MicroBlaze processor design on the Zynq All Programmable SoC.

This is to prevent the tools from automatically trying to configure the underlying Zynq AP SoC platform, even though a MicroBlaze processor is targeted.

3-6-6. Click **Run** to close the dialog box and launch the software application on the hardware evaluation board (4).

3-7. View application output in the terminal.

3-7-1. Select the **Terminal** tab to view the application output.

You should observe the "SDKintro test code starting..." string output in the terminal.

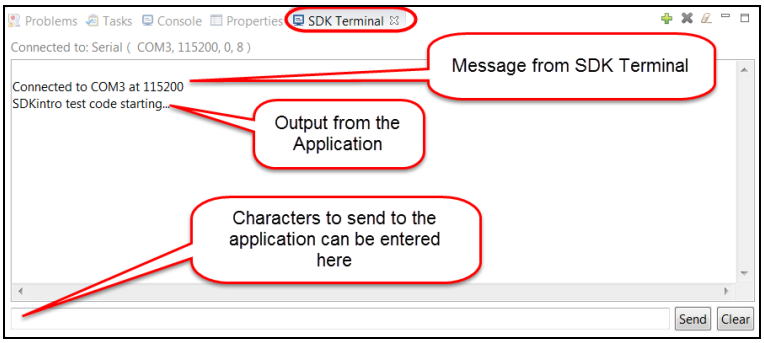


Figure 3-25: Viewing the Output in the Terminal

3-8. Instruct the program to drive the LEDs

3-8-1. Enter in the character "+" at the base of the SDK Terminal view and click the **Send** button in the SDK Terminal view, located to the right side of the text entry, to send that character to the application running on the development board.

This will instruct the application to drive the LEDs with a gaussian display pattern.

3-8-2. View the User LEDs on the development board.

Note: The LEDs will be turned on and each will have a different intensity, decreasing from high to low. Setting the intensity of the LEDs is handled by the application. The LEDs are connected to pins in the PL and the only way to drive them through software is to interface with an associated IP Block instantiated in the PL.

Question 12

Where does the *writeStr_NL()* routine output to? Where are the input (stdin) and output (stdout) devices configured?

Question 13

If the software was modified, would it be necessary to reimplement (translate, map, place & route) the hardware? What would need to happen if the software was modified? Describe the relationship between hardware and software updates.

3-9. Close the Xilinx SDK tool.

3-9-1. Select **File** > **Exit** to save the SDK configuration and exit the tool.

If you have not configured your workspace to not request confirmation on close, a dialog box will appear confirming that you wish to close SDK.

You have the option to set SDK to never ask you for exit confirmation.

3-9-2. Click **Yes** if this dialog box appears to conclude the saving of the SDK configuration and exiting of the tool.

Summary

You have now walked through all of the high-level steps required for building an embedded software project:

- Creating the hardware, BSP, and application project
- Adding existing source code to the application project
- Locating and correcting syntax errors
- Running the application on hardware

Answers

1. What action might you take to resolve this issue?

   This issue can be resolved by defining a variable named *delay* within the function (prior to when it is used).

2. What is the best way to investigate and correct this error?

   Since there is a warning icon immediately prior to this error, it is logical to assume that there might be a connection. The warning indicates that the variable *dely* is not used. It is reasonable to assume that this is simply a typographical error and that *dely* should really be *delay*.

3. Unfortunately, the default for the SDK editor is to have line numbering turned off. How can you turn on line numbering?

   Right-click in the left margin of the editor pane amd select **Show Line Numbers** from the context menu.

4. What happened as a result of the recompilation of the file? **Hint:** Look at the Problems tab.

   The correct spelling of the variable name made the error in *LED_drivers.c* go away. This exposes another error in another file.

5. What do you think these small red and yellow rectangles mean?

   These are markers that correspond to the locations of warnings and errors in the source file. Yellow represents warnings while red represents errors. Clicking a red rectangle changes the source code view to present the line of code corresponding to that error.

6. What happens as soon as you add the closing parenthesis?

   Certain types of errors and warnings are quickly detectable without having to recompile the code. As soon as the closing parenthesis, the tools detect the change and rescan the source code. With the conditional statement now complete, many of the warnings disappear from the error and navigation margin.

7. How do you use the Outline view?

   Click the element listed in the Outline view to jump to its location in the source file.

8. What line number does the *LED_driver_select(int)* function begin on? (Located in LED_drivers.c)

   On or about line 64.

9. What happens if you click a different source tab?

   The Outline tab changes to match the active source tab.

10. What is the easiest way to open *utils_print.h*?

    While you could return to the Project Explorer and locate *utils_print.h* and double-click it, it is far easier to press-and-hold <**Ctrl**> and hover the mouse over *utils_print.h*. This will turn the *include* into a hyperlink. To open the *include* file in a new window, just click the hyperlinked *include* file.

11. What else turns into a hyperlink?

    Press and hold <**Ctrl**> and hover it over a number of different elements. You will find that everything that is listed in the Outline tab is hyperlinked. This is particularly useful for quickly locating the source for functions, macro expansions, etc.

12. Where does the *writeStr_NL()* routine output to? Where are the input (stdin) and output (stdout) devices configured?

    The *writeStr_NL()* routine outputs to the stdout, which is mapped to the serial port output device. Similarly, the stdin is also mapped to the serial port input device.

    These mappings are made in the Board Support Package settings as shown in the figure below (the BSP for the Zynq platform is shown in the image below. If it were the BSP for the MicroBlaze, the *Value* would be axi_uartlite_0).

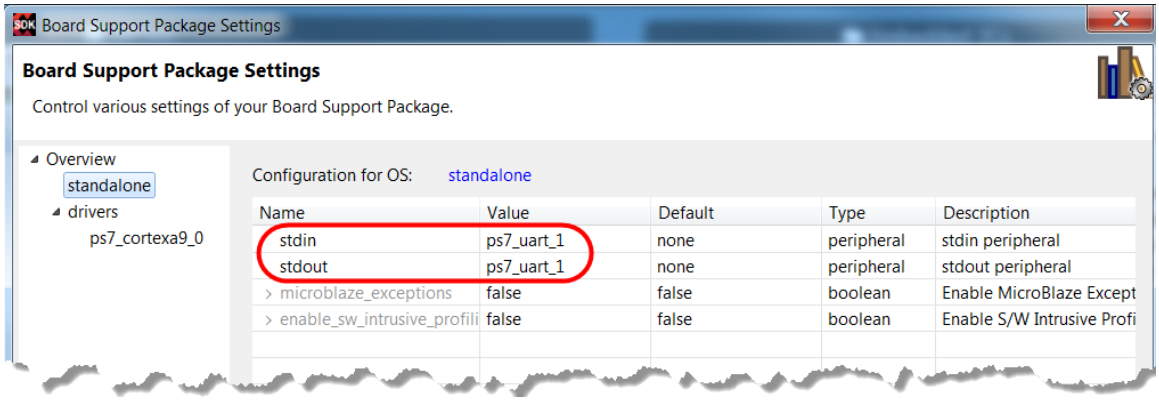


Figure 3-26: Mapping STDIO in the BSP Settings

13. If the software was modified, would it be necessary to reimplement (translate, map, place & route) the hardware? What would need to happen if the software was modified? Describe the relationship between hardware and software updates.

    The dependency runs from hardware to software, which means that changes in software will not affect hardware. However, changes in hardware (specifically the HDF file) will be detected by SDK and it will rebuild the hardware platform which will, in turn, cause the rebuilding of the BSP.

    Rebuilding of the BSP might impact the application. Consider if a peripheral were being accessed in software, then the peripheral were removed from the design. This would result in numerous software errors in which attempts were made to reference a non-existent peripheral.

Lab 4: System Debugger

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

This lab shows you how to use the basic features of the System Debugger, including a number of execution controls and data monitoring and modification techniques.

Objectives

After completing this lab you will be able to:

- Control execution by using single-stepping techniques and breakpoints
- Monitor and modify variables and regions in memory

Introduction

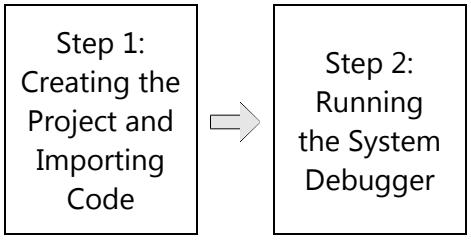
The Towers of Hanoi game is a puzzle consisting of a number of disks sorted from smallest (on the top of the stack of disks) to the largest (on the bottom of the stack) and three vertical rods that are used to hold the disks.

The game is played by moving the entire stack of disks from one vertical rod to another while following three simple rules:

- Only one disk can be moved at a time.
- Only the topmost disk on the stack can be moved to the topmost position on either of the other rods.
- No disk can be placed on a smaller disk.

While there are a number of algorithms that can be employed to solve this puzzle, the provided code uses a recursive algorithm. Recursion is the process in which a function calls itself. By the very nature of recursion, the call stack is used heavily.

General Flow



Creating the SDK Project and Importing Source Code Step 1

You will begin this lab by creating a set of SDK projects and import the source code for a C application that you will need to verify for proper behavior.

You will use *C:\training\systemDebugger\lab* as the SDK tool workspace. The development environment will automatically create a hardware platform based on the development board selected.

Several common and repetitive tasks have been automated with a custom tool called *SDK Launcher*. This tool will automatically create an SDK workspace, create a hardware platform specification targeting the desired development board and processor combination and, finally, launch the SDK tool in the open workspace.

The hardware platform specification will be named *UED_hw*. If you are unsure how to do any of these tasks manually, reference the relevant sections in the *Lab Reference Guide*.

1-1. Open the SDK Launcher tool.

1-1-1. Open Windows Explorer and browse to the *C:\training\tools* directory.

1-1-2. Double-click the **SDKlauncher.jar** file to launch the application.

1-2. Configure the SDK Launcher.

1-2-1. Click the Browse icon button to open a file browser window (1).

1-2-2. Select *C:\training\systemDebugger* to identify which lab to work with.

1-2-3. Click **Open** to select the path and return to the tool's main GUI.

1-2-4. Click the Down arrow next to the platform list to expand the list (2).

1-2-5. Select the **[PS | MicroBlaze on ZC702 | ZedBoard].**

Note: Selecting PS will target the ARM® processor and MicroBlaze will target the MicroBlaze™ processor.

1-2-6. Ensure that Vivado version **2016.3** is populated in the SDK Version field (3).

This instructs the SDK Launcher application to use a particular version of the SDK tool.

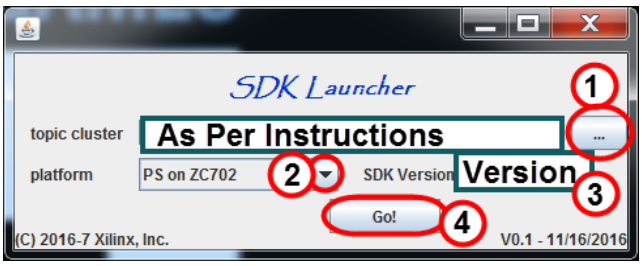


Figure 4-1: Configuring the SDK Launcher Tool

1-2-7. Click **Go** to automatically set up the workspace, create the hardware platform specification, and launch SDK (4).

Close the Welcome screen once SDK launches, if it is open.

Using the Application Project Wizard is a quick way to set up a C or C++ software application project that targets an existing processor and OS platform (Standalone or Linux). You can automatically generate the board support package (BSP) or select an existing one. Based on the dialog box choices, the appropriate tool chain is selected for pre-processing, compiling, assembling, and linking.

1-3. Create a new C/C++ application project named *debugger_app*. Use the board support package named *debugger_bsp*.

1-3-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

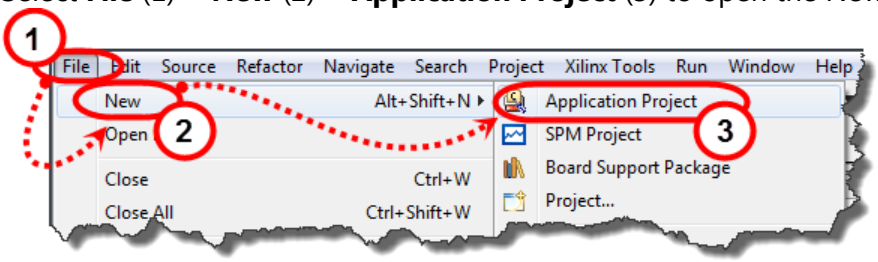


Figure 4-2: Creating an Application Project

1-3-2. Enter **debugger_app** as the project name.

1-3-3. Ensure that you have **UED_hw** selected from the Hardware Platform drop-down list as the SDK tool can manage multiple platforms within a single workspace.

This will populate the Processor drop-down list accordingly.

1-3-4. Ensure that **ps7_cortexa9_0 if using the PS or microblaze_0** is selected from the Processor drop-down list.

1-3-5. Ensure that **standalone** is selected from the OS Platform drop-down list.

1-3-6. Select **Use Existing** and choose an existing BSP from the drop-down list if you have already created a BSP for this hardware platform; otherwise, select **Create New** and enter the name for the BSP as **debugger_bsp**.

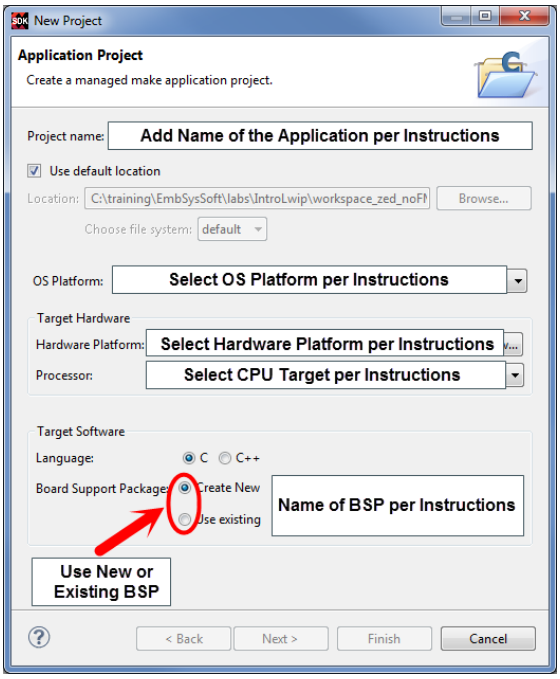


Figure 4-3: Entering Application Project Information

1-3-7. Click **Next** to select the template for this application.

1-3-8. Select **Empty Application** (1).

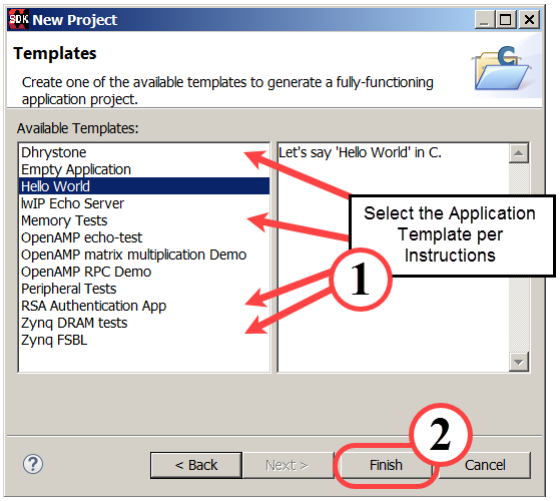


Figure 4-4: Selecting an Application Template (Selection in Figure May Not Match Instructions)

1-3-9. Click **Finish** to create the new project (2).

As the project is created, the new BSP is compiled and any sources from the templates are also compiled. The creation of the new project usually takes less than a minute.

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

1-4. Add *all source (*.c) and header (*.h) files* to the application.

The preferred method for importing sources is shown here.

1-4-1. Expand the project named **debugger_app** > **src** using the Project Explorer.

1-4-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

1-4-3. Select **Import** to open the Import Wizard (2).

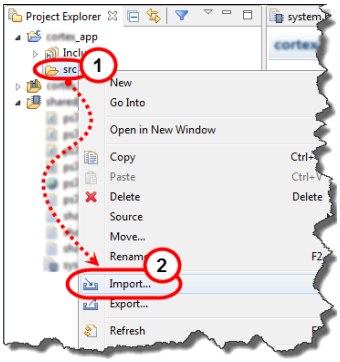


Figure 4-5: Importing a Resource File

The Import Wizard dialog box opens.

1-4-4. Expand **General** (1).

1-4-5. Select **File System** as you will be selecting individual files directly from the file system (2).

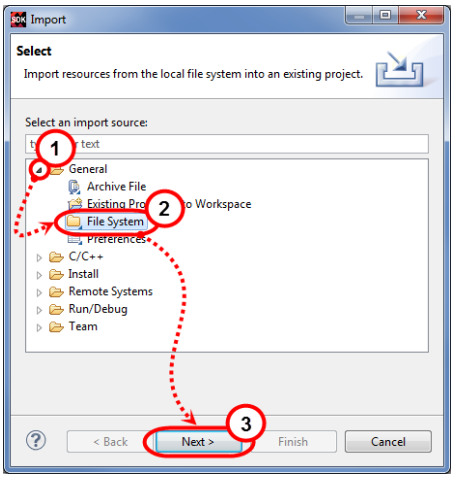


Figure 4-6: Selecting File System

1-4-6. Click **Next** to advance to specifying the files to import (3).

1-4-7. Browse to *C:\training\systemDebugger\support\src* in the *From directory* field.

1-4-8. Select the file(s) by checking the box beside **all source (*.c) and header (*.h) files**.

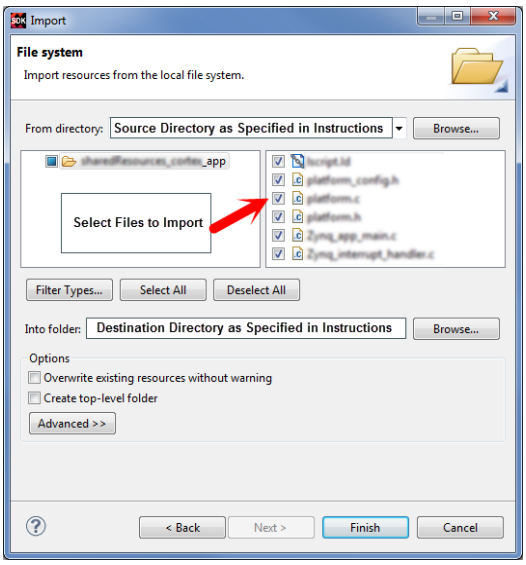


Figure 4-7: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

1-4-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

1-5. Build all of the projects.

1-5-1. Force the building of all applications by selecting **Project** > **Build All**.

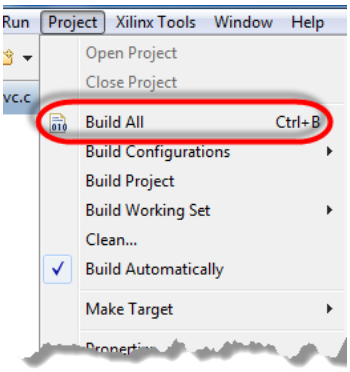


Figure 4-8: Selecting Build All

1-6. Apply power to the board.

1-6-1. Switch the board to OFF to clear memory if it is powered on and has a power slide switch; otherwise, remove the power cable.

1-6-2. Ensure that you have a correctly imaged SD card if you are using the MicroBlaze Processor.

The correct SD card images are available from *C:\training\systemDebugger\support\TransUartPins48-49_BootImage\[ZC702 | ZED]*. Copy the BOOT.bin to the root of the SD card.

1-6-3. **MicroBlaze processor users only:** Insert the SD card into the development board. This will configure the PS with a special application to allow UART communications with the MicroBlaze through the PS.

Note: You may have to change the jumper/switch settings to support JTAG programming. Many devices, such as the Zynq®-7000 family support JTAG fallback if the jumpers/switches are set to SD card and no card is present. Other families, such as the Zynq UltraScale+™ family, need to be specifically configured to boot from JTAG or SD Card. Refer to the lab setup guide for your board for details.

1-6-4. Slide the switch to the ON position if your board has a power switch (as shown in the lab setup guide for your board); otherwise, plug the power cable in.

Important note: ZC702 board users can continue with the instructions below to make a serial connection. ZedBoard users should skip to the instruction 1-9 below that begins with "Launch the Tera Term terminal program" as there are issues between the SDK tool and the Cypress USB-to-serial driver.

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

1-7. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

1-7-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

1-8. Configure the SDK Terminal.

1-8-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

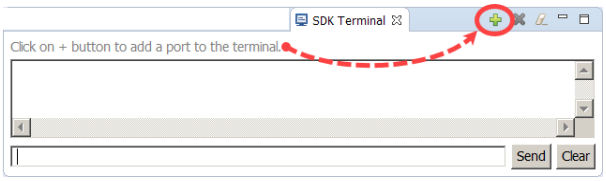


Figure 4-9: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

1-8-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

1-8-3. Set the baud rate to **115200** (2).

1-8-4. Leave the other settings at their defaults.

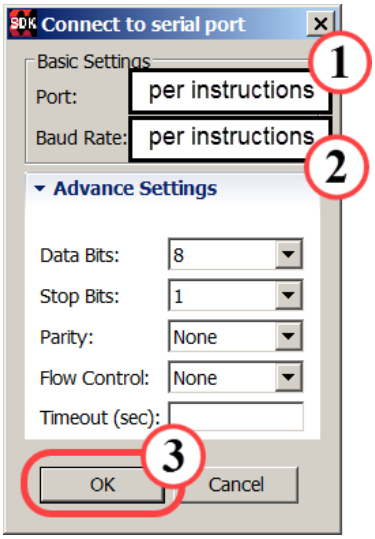


Figure 4-10: Configuring the SDK Terminal

1-8-5. Click **OK** to save these settings and begin the terminal session (3).

Important note: ZC702 board users can skip this next instruction. ZedBoard users will need perform this instruction to successfully circumvent the issues between the SDK tool and the Cypress USB-to-serial driver.

Tera Term is a popular public domain terminal emulation program. It is capable of operating as a serial port terminal or as a telnet client. It is an alternative to using the Terminal view that is built into SDK.

1-9. Launch the Tera Term terminal program.

1-9-1. Double-click the **Tera Term** icon on the Windows desktop to launch Tera Term.

Alternatively, you can select **Start** > **All Programs** > **Tera Term** > **Tera Term**.

1-9-2. Select **Serial** as the connection (1).

1-9-3. Click the **Port** drop-down list to view the available COM ports (2).

Note: If your port is not listed, exit Tera Term, power cycle your board and re-start this step.

1-9-4. Select the COM **#** discovered in the last step (3).

Note that the ZC702 will show the Silicon Labs driver as shown in the figure below and the ZedBoard will show the Cypress driver or possibly just a USB Serial Port entry.

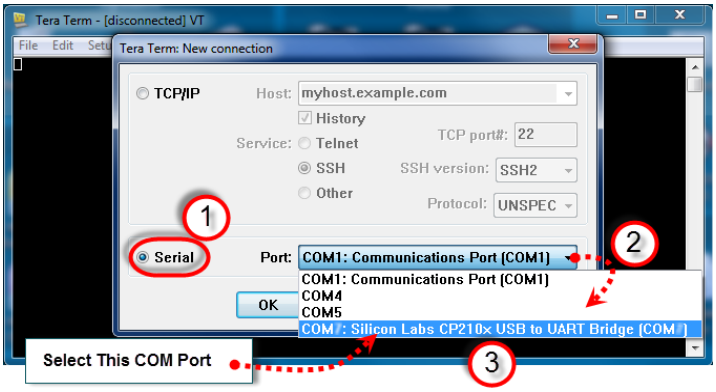


Figure 4-11: Selecting the COM Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

1-9-5. Click **OK**.

The terminal console window opens.

1-9-6. Select **Setup** > **Serial Port**.

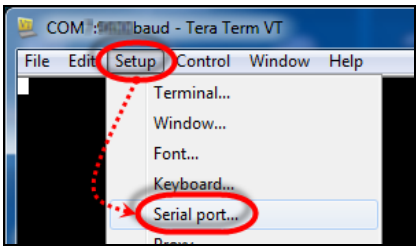


Figure 4-12: Opening the Tera Term Serial Port Setup Window

The Tera Term Serial Port Setup dialog box opens.

1-9-7. Confirm that the proper serial port has been selected (1).

1-9-8. Set the baud rate to **115200** (2).

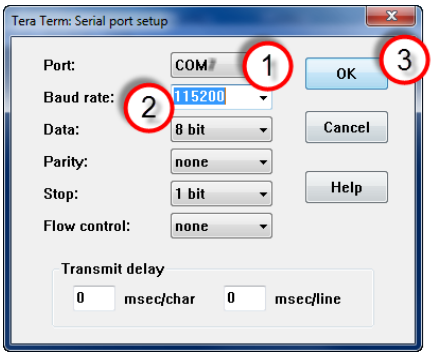


Figure 4-13: Setting the Parameters for the Serial Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

1-9-9. Click **OK** (3).

Tera Term is now configured to receive and transmit serial information to/from the evaluation board.

While there are a number of source files in the application project (*hanoiTest_app*), there is only one that need concern you now and that is the *Towers_of_Hanoi.c* file.

1-10. Open *Towers_of_Hanoi.c* in the editor.

1-10-1. Locate **Towers_of_Hanoi.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

1-10-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

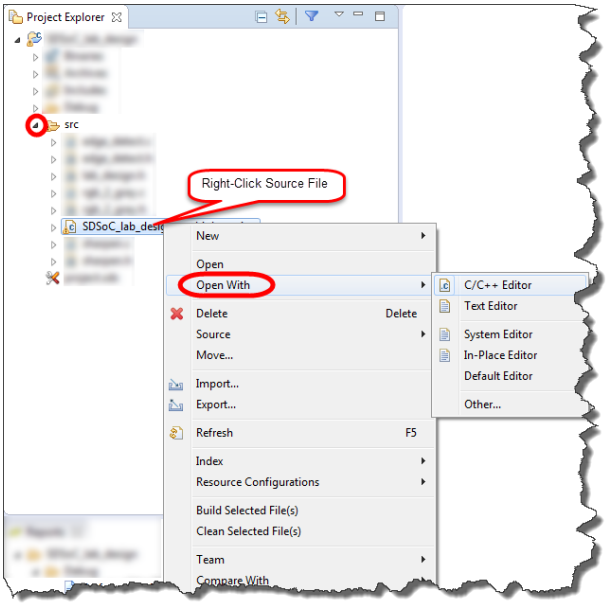


Figure 4-14: Opening a Source File via Right-Click

Line numbering is off by default, but it can be easily turned on.

1-11. Turn on line numbering in the text editor.

1-11-1. Open the source file in the editor if it is not already open.

1-11-2. Right-click in the left margin of the text editor.

1-11-3. Select **Show Line Numbers**.

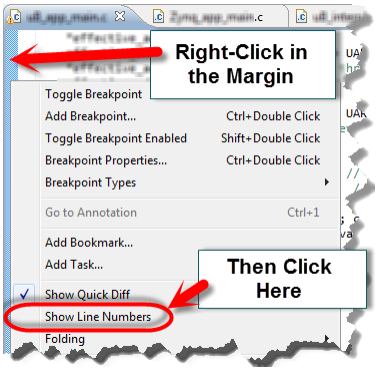


Figure 4-15: Enabling Line Numbering

Running the System Debugger Step 2

Note: The hardware design used for this lab contains some Programming Logic (PL) components that must be programmed. Both the ARM and MicroBlaze designs require the PL to be programmed.

2-1. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

2-1-1. Select **Xilinx Tools** > **Program FPGA** or click the icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

2-1-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

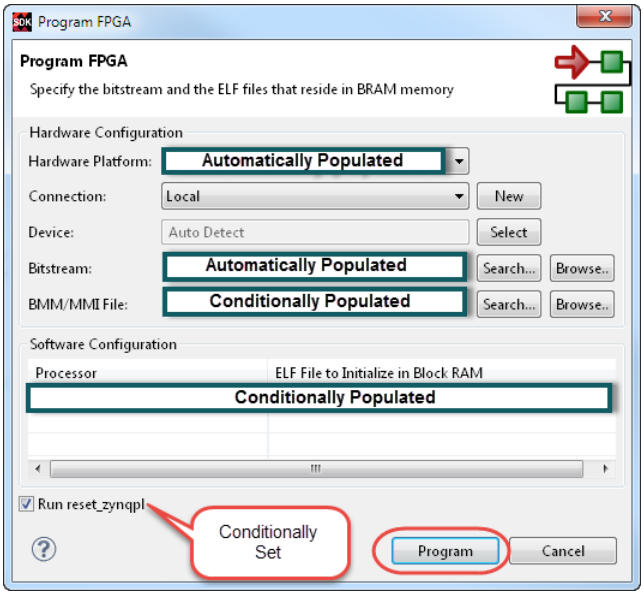


Figure 4-16: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

If you are programming a MicroBlaze™ softcore processor on a Zed or ZC702 Development Board a warning indicating that there is not a PS in the design will be presented. Read the warning and click OK to dismiss it. The design does not use any PS managed or initialized resources.

With the SDK environment configured, you will now enter SDK's debugging mode in which you can both control and observe the behavior of the code.

A Debug configuration defines how you want the system to work when performing a debug operation and maps an ELF object file to a target for execution. Typically, a debug operation switches to the Debug perspective. While there are a significant number of switches and options, the most common are shown below.

2-2. Set up a debug configuration for a specific application project.

2-2-1. Right-click the application project that you want to build the Debug configuration for from the Project Explorer pane (1).

2-2-2. Select **Debug As** to open the menu of predefined configurations and the configuration manager (2).

2-2-3. Select **Debug Configurations** to view all the available debug options (3).

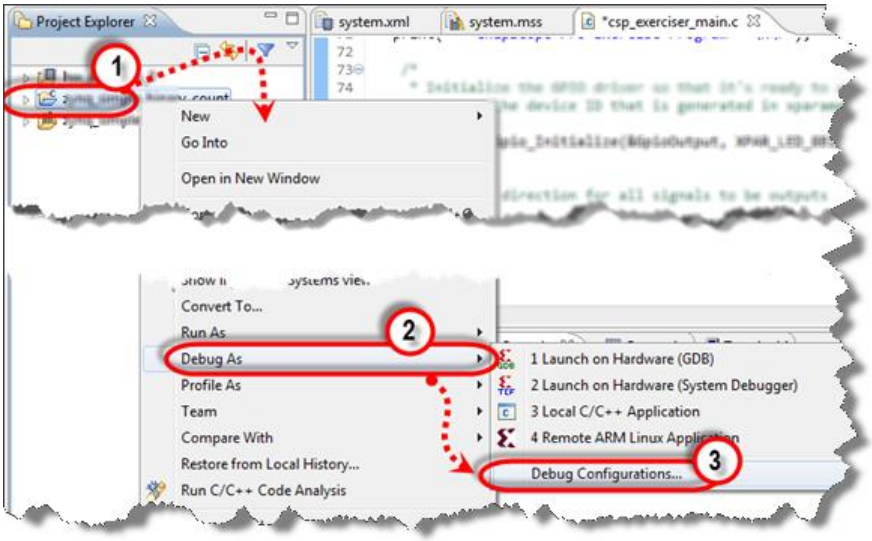


Figure 4-17: Creating a Debug Configuration

The Debug Configurations dialog box opens.

2-2-4. Double-click **Xilinx C/C++ application (System Debugger)** to create a launch configuration (1).

Alternatively, you can select **Xilinx C/C++ application (System Debugger)** and click the **New** configuration button.

If this is the first debug configuration being created for the project, a welcome type dialog opens. If one or more configurations exist, then the last open configuration will be displayed. In either case, a new configuration can always be added. Existing configurations are shown in the left pane and can be selected for debugging.

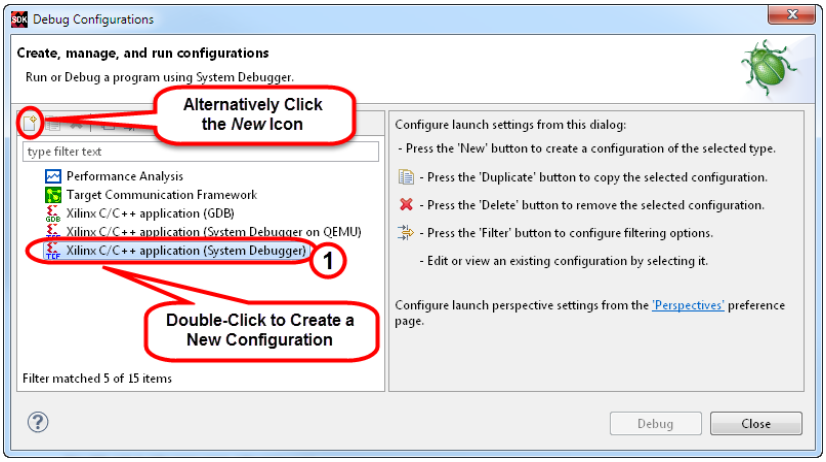


Figure 4-18: Creating a New System Run/Debug Configuration

A new Debug Configuration is created named *System Debugger using Debug_debugger_app.elf on Local* (2).

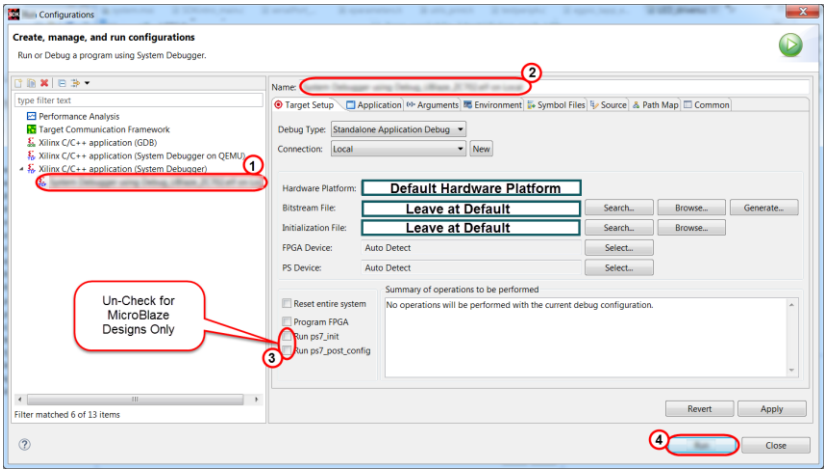


Figure 4-19: Run/Debug Configurations Dialog Box (Zynq AP SoC)

The new configuration will appear with other existing configurations and have the name of your application. You will also note that a number of fields are automatically filled in for you using the name of your application as the basis (that is, if your application is named "XYZ", then the Name field will be populated with XYZ Debug.) Most users will leave the other settings at their default.

2-2-5. Deselect the **Run ps7_init** and **Run ps7_post_config** options (3) if you are targeting a MicroBlaze processor design on the Zynq All Programmable SoC.

This is to prevent the tools from automatically trying to configure the underlying Zynq AP SoC platform, even though a MicroBlaze processor is targeted.

2-2-6. Click **Debug** to close the dialog box and launch the software application on the hardware evaluation board (4).

If the Confirm Perspective Switch dialog box appears, click **Yes**.

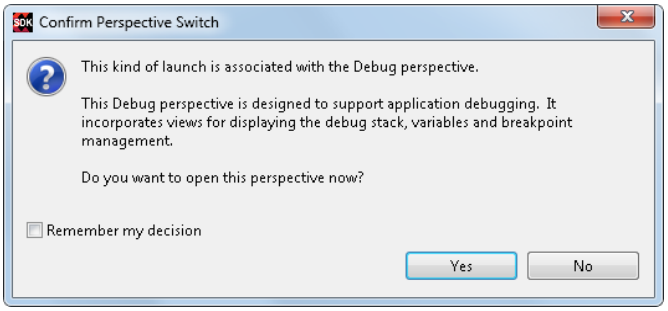


Figure 4-20: Confirming Switch of Perspective

The Debug Perspective view opens.

You will now use the debugger to perform the two basic tasks of debugging: controlling execution and inspecting and modifying memory.

Question 1

At this point is the application running or halted? If it is halted, where and why?

Place a breakpoint at the portion of code where the number of disks is retrieved from the user (on or about line 128). The user's entry is converted from a string to a number using the function *strtol*. This allows the code to perform any initializations and lets you start getting into the algorithmic portion of the code without having to perform a lot of single steps.

2-3. Find *strtol(*.

The find/replace operation is accessible through both a click sequence and a keyboard shortcut.

2-3-1. Select **Edit** > **Find/Replace** or press <**Ctrl** + **F**>.

The Find/Replace dialog box opens.

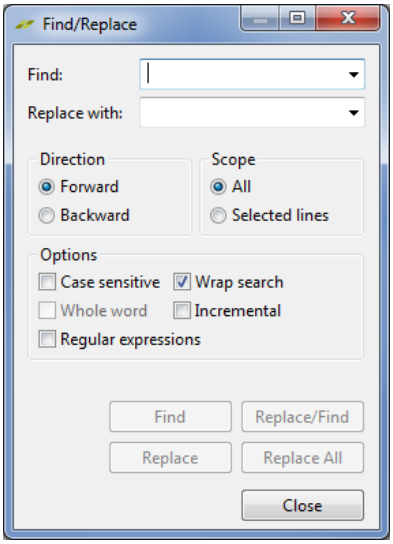


Figure 4-21: Default Find/Replace Dialog Box

2-3-2. Enter **strtol(** in the Find field.

2-3-3. Click **Find** or press <**Enter**> to find the next occurrence of *strtol(*.

2-3-4. Continue clicking **Find** or pressing <**Enter**> until you locate the specific instance that you are looking for.

With the **Wrap search** option enabled, the file is treated as a continuous loop and the find operation will jump to the next occurrence at the top of the file (when searching forwards) or at the bottom of the file (when searching backwards). A "ding" sound is made when the search wraps around.

If you are looking for text within a specific region of the code you would first highlight the region to perform the search in, then launch the Find/Replace function as described in this topic.

2-3-5. Click **Close** to close the Find/Replace dialog box.

Since you are now preparing to set the unconditional breakpoint, open the Breakpoints view so that you can see all the existing breakpoints as well as the new breakpoint that you will create.

2-4. Open the Breakpoints view (tab).

2-4-1. Select the **Breakpoints** tab to see all of the automatically generated breakpoints as well as any breakpoints that you have created.

Tip: If you click the **X** in the tab, the tab will close. To reopen the view, select **Window** > **Show View** and select the view that you want to open.

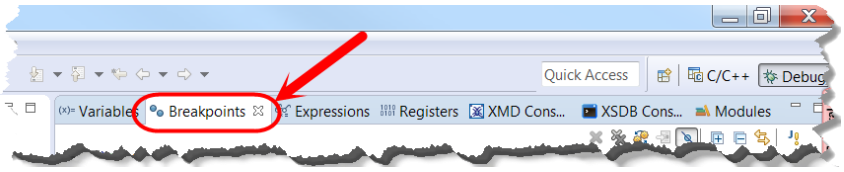


Figure 4-22: Locating the Breakpoints Tab

2-5. Add a breakpoint at the current line.

2-5-1. Right-click the left margin in the editor on the line containing the function call to strtol.

2-5-2. Select **Add Breakpoint** or **Toggle Breakpoint**.

If you select Add Breakpoint, then the Breakpoint Properties dialog box opens. Here you can specify the details for this breakpoint if necessary to create a conditional breakpoint.

If you select Toggle Breakpoint, then a simple breakpoint will be created (or removed if one already existed) on the line you specified using default breakpoint properties.

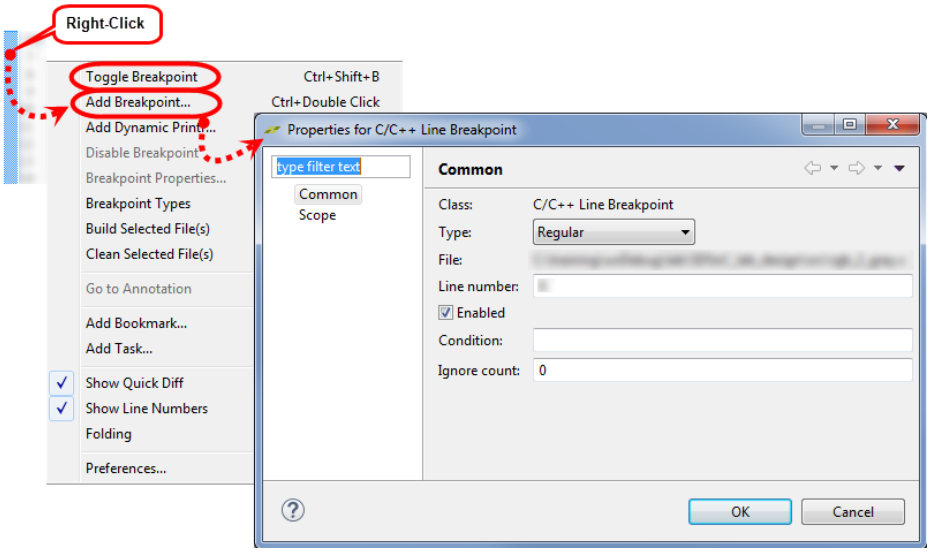


Figure 4-23: Adding a Simple or Conditional Breakpoint

2-5-3. Click **OK** to create the breakpoint if Add Breakpoint is selected.

2-6. Run to the next breakpoint.

2-6-1. Click the **Resume** icon ( ) or press <**F8**> to continue operation.

The application will run until:

- An unconditional breakpoint is met,
- A conditional breakpoint whose condition is satisfied is met, or
- The user halts ( ) or pauses ( ) execution

Notice that in the SDK Terminal or the Tera Term Terminal several messages have printed out. One is the welcome message from the application, and another is a request for you to enter the number of disks you want to solve for in this run.

2-7. Run a two-disk solution.

2-7-1. Type the following into the terminal (this is NOT a misprint):

```
3 <Enter>
```

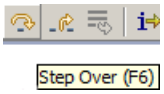
Notice that the program execution has halted which is indicated by the green highlighting of the line containing the breakpoint.

Since the function *strtol()* is known to be working correctly, you can step over it.

2-8. Step OVER the *strtol()* function.

2-8-1. Click the **Step Over** ( ) icon or press <**F6**>.

The Step Over icon can be found in the icon bar below the Run entry in the menu bar.



Wait a minute—didn't you want to run only a two-disk solution? But you entered a 3!

Question 2

How can you "go back" and ask for only a two disk solution?

2-9. Change the value of the variable *number_of_disks* from 3 to 2.

2-9-1. Select the **Variables** tab.

This tab typically appears in the upper-right window. If it is not available, select **Windows** > **Show View** > **Variables**.

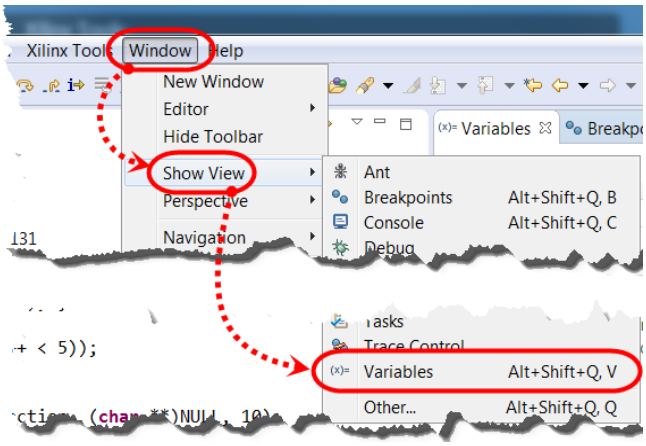


Figure 4-24: Making the Variables Tab Visible

2-9-2. Click in the cell on the row named *number_of_disks* in the Value column.

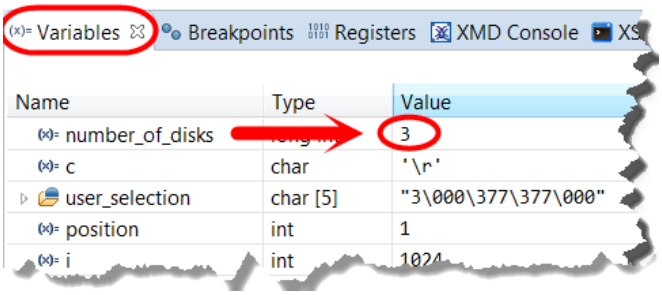


Figure 4-25: Selecting the Variable Value to Change

2-9-3. Type in the following:

```
2 <Enter>
```

You have now successfully changed the value of *number_of_disks* to 2.

2-10. Run the code up to the point where the *do_hanoi()* function is called.

2-10-1. Locate the function *do_hanoi* using the search method shown above.

Note that this is the *call* to *do_hanoi()* in the *main()*, not the function prototype or function definition.

2-10-2. Set a breakpoint on that line.

2-10-3. Click the **Resume** ( ) icon or press <**F8**> to run the code to that point.

2-10-4. Type the following when the application asks to show the solution:

```
Y <Enter>
```

Note that Tera Term does not require the <Enter>. The SDK terminal required the <Enter> or clicking the Send button.

This option shows each individual step used to solve the Towers of Hanoi problem by illustrating from which tower the disk is pulled from to where it is deposited.

Sometimes, for very large numbers of disks, the output to the serial port can significantly slow the execution of the code. This is where the "N" option is worthwhile as it will tell you the number of steps required to solve the puzzle, but not the steps themselves. This feature is used to illustrate how stack overflow situations can be debugged.

2-11. Step into the *do_hanoi()* function to inspect the code.

2-11-1. Click the **Step Into** () icon or press <**F5**>.

The first line of code is a protection against a zero or negative number of disks.

2-11-2. Use either the **Step Into** or **Step Over** icons to execute this line of code.

The next line of code is the first recursive call to the *do_hanoi()* function.

Notice the current debug stack (you will need to note the call stack in order to answer the next question).

2-11-3. Click the **Step Into** () to enter the first recursive call of *do_hanoi*.

Question 3

What changed in the Debug (call stack) window?

Question 4

What changed in the Variables tab?

Question 5

Should you continue stepping into or stepping over this function?

2-11-4. Click the **Resume** () icon or press <**F8**> to finish executing this function.

The code continues to execute and displays the solution in the SDK Terminal window.

Question 6

Do you agree with the solution for the two disk problem?

Question 7

What are your choices regarding stopping at a point deep inside a recursive function (or loop)?

Now consider the situation where the number of disks is large and you want to see what is happening at the end of the recursive processes.

2-12. Set a conditional breakpoint on the first recursive call to *do_hanoi()* so that the breakpoint becomes active when the value of N is less than 3.

2-12-1. Locate the line (near line 179) which reads:

```
do_hanoi(N-1, from, using, to, show_solution);
```

2-12-2. Right-click in the left margin to open the context menu.

2-12-3. Select **Add Breakpoint** (rather than Toggle Breakpoint).

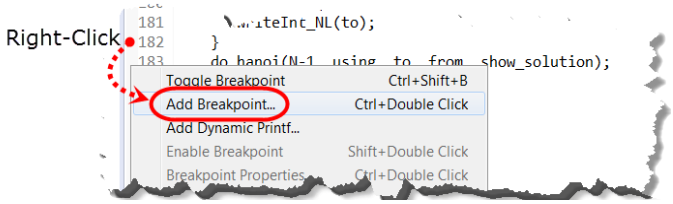


Figure 4-26: Adding a Conditional Breakpoint

The Breakpoint Dialog box opens.

2-12-4. Set the **Condition** field to `N < 3` (spaces are ignored).

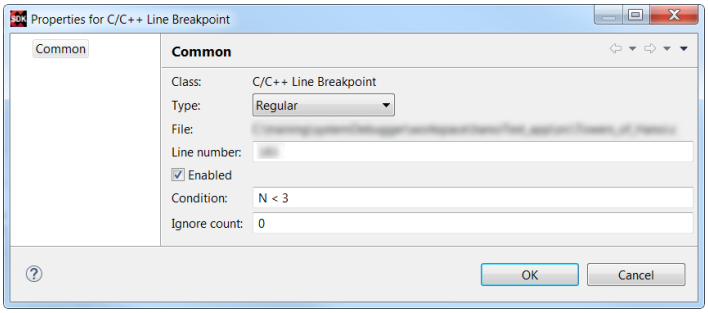


Figure 4-27: Configuring the Conditional Breakpoint

2-12-5. Click **OK** to set the conditional breakpoint.

Note: This breakpoint's conditions can be changed by using the Breakpoint Properties entry in the pull-down menu or by right-clicking the breakpoint in the Breakpoint window and selecting **Breakpoint Properties**.

Question 8

How can you tell if a breakpoint is conditional or not?

2-13. Disable all other breakpoints other than the conditional breakpoint that you just set.

2-13-1. Select the **Breakpoints** tab.

2-13-2. Uncheck all of the breakpoints other than the conditional breakpoint and the [function: main] breakpoint and the [function: _exit] breakpoint.

If you uncheck the [function: main] breakpoint, the next time you restart the debug session, the code will run until one of your active breakpoints is reached. It will not stop at the first executable line of code.

Note: This process disables the breakpoint, but it does not remove them. You can permanently remove a breakpoint by selecting the breakpoint in the Breakpoints window and clicking the 'X' icon (remove breakpoint).

2-14. Run another data set through the application by using 12 disks.

2-14-1. Enter the following into the SDK terminal:

```
12 <Enter>
Y <Enter>
```

The code is executed and stops when N becomes less than three in the *do_hanoi()* function.

Question 9

Did the conditional breakpoint work?

_____

_____

_____

2-15. Continue running the code, skipping the remainder of the breakpoints.

2-15-1. Click the **Skip All Breakpoints** ( ) icon to instruct the debugger to ignore the remaining breakpoints.

2-15-2. Click the **Resume** icon ( ) to run the application to the beginning of the next data set.

You have probably stumbled across a variable called *buffer_space*. This array is used to detect stack overflow. While learning how to detect stack overflows is not the focus of this lab, you will use this variable to practice monitoring memory locations.

A brief explanation of how this is used to detect stack overflows: The way the linker script is written, global variables sit adjacent (and in lower memory) to the stack. Therefore, if the stack overflows (stacks grow toward lower memory) it will begin to overwrite the values in this buffer.

To test for stack overflows, the buffer is filled with a known value prior to the stack-intensive function *do_hanoi()* being called. After the call to *do_hanoi()* is made, the buffer is scanned for any changes in value. If a change is detected, it will be due to the stack overflow.

From a practical standpoint, this is easily remedied by either putting limits on how many disks can be used in the simulation, and/or enlarging the stack, which is accomplished by modifying the linker script and is the subject of another lab.

2-16. Find the memory location of the array variable *buffer_space* and monitor that memory location.

Since the variable *buffer_space* is located in global memory, it cannot be seen by using the Variables tab.

2-16-1. Select **Window** > **Show View** > **Expressions** to make the Expressions tab visible.

2-16-2. Click the green **Add New Expression** icon to define the new expression to monitor.

2-16-3. Type the following:

```
buffer_space <Enter>
```

2-16-4. Click the entry for *buffer_space* to see the pertinent information below the expressions table.

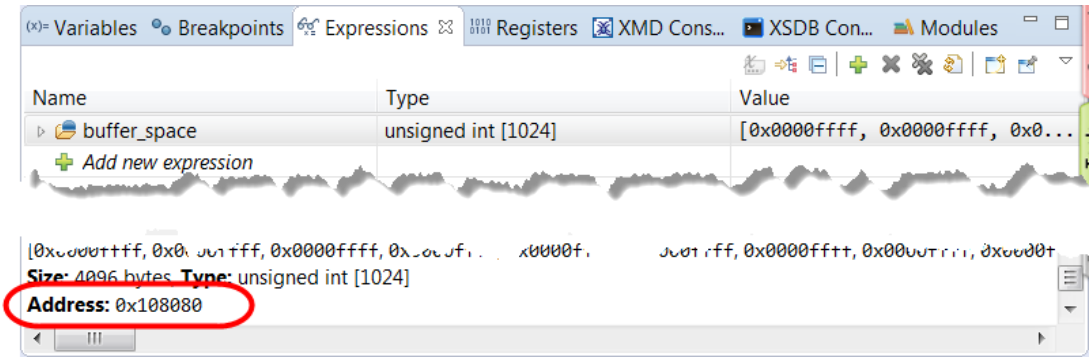


Figure 4-28: Locating the Address of a Variable in the Expressions Window

Note: Your address may differ.

2-16-5. Note the memory address of the variable as you will be using it shortly.

Note: The address is in hex and begins with 0x. You may need to scroll down to see the address or expand the window. You could have also retrieved this information by hovering the mouse over the *buffer_space* variable name in the source code.

2-16-6. Select the **Memory** tab.

The Memory tab is typically located in the same window as the Console tab, but is sometimes found in the same window as the Breakpoints tab. Remember that these tabs can be relocated for your convenience.

If the Memory tab is not available, select **Window** > **Show View** > **Memory**.

2-16-7. Click the green '✚' sign in the Memory tab toolbar to add a memory monitor.

The Memory Monitor dialog box appears, requesting the address to be monitored.

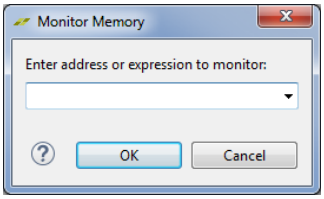


Figure 4-29: Monitor Memory Dialog Box

2-16-8. Enter the hex address that you previously obtained for the *buffer_space* variable into the dialog box.

Remember to type the address as a hex value by using the *0x* prefix.

2-16-9. Click **OK** to begin monitoring this location in memory.

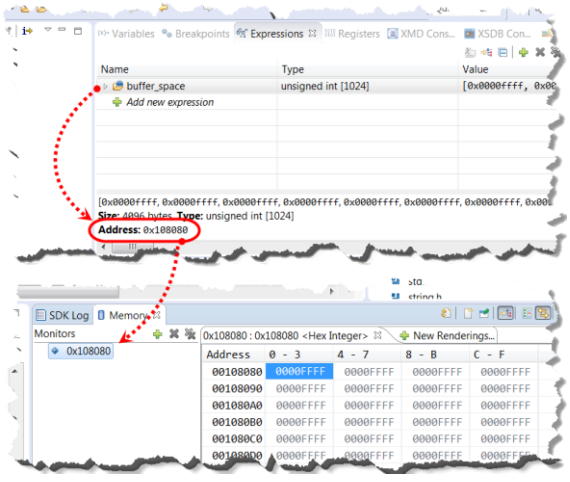


Figure 4-30: Looking at buffer_space in Memory

As can be seen in the Memory window, the value at address 0x108080 has a hex value of 0x0000FFFF, which is equivalent to 65537 decimal.

Note: Your address values may not be exactly the same as in this example.

2-17. Modify the value of a memory location.

This will inject a deliberate error into the *buffer_space* variable and should be detected and reported when the code is run.

2-17-1. Double-click the cell at the intersection of the fourth row beneath the buffer space address and 4-7 with the Memory window still open.

This is an arbitrary location within the buffer_space.

2-17-2. Enter the arbitrary value **80** as shown below.

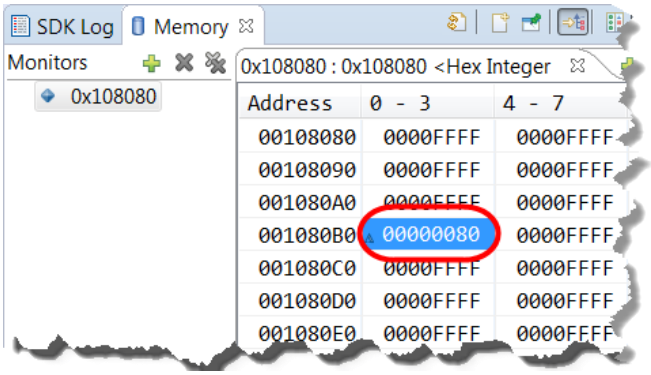


Figure 4-31: Changing the Value of a Cell within the buffer_space Array

2-17-3. Press <**Enter**> to commit this value to the array.

2-18. Run another 12 disk simulation to test the behavior of the changes made to the *buffer_space* array.

2-18-1. Enter **12** as the number of disks.

2-18-2. Enter **N** for Show solution.

This will make the simulation run much faster as there is little serial output to slow the processor down.

Question 10

Was an incursion into the buffer space found?

2-19. Close the Xilinx SDK tool.

2-19-1. Select **File** > **Exit** to save the SDK configuration and exit the tool.

If you have not configured your workspace to not request confirmation on close, a dialog box will appear confirming that you wish to close SDK.

You have the option to set SDK to never ask you for exit confirmation.

Click **Yes** if this dialog box appears to conclude the saving of the SDK configuration and exiting of the tool.

Summary

You have just completed the basic introduction to the System Debugger tool. You controlled execution through single-stepping and setting breakpoints and monitored and changed variables and memory.

While there are some additional features available, this lab covered enough capabilities of the tool for you to be successful in debugging your software applications.

Answers

1. At this point is the application running or halted? If it is halted, where and why?

   The debugger automatically runs the code until a breakpoint is reached. As part of the debug configuration, a breakpoint is automatically inserted on the first line of executable user code. By the time you begin interacting with the debugger, you will see that the execution has paused at the first line of your code.

   When applications are running, none of the variables, expressions, memory, etc. are visible. When the application is paused or suspended, a green bar appears on the next line of code to be executed.

2. How can you "go back" and ask for only a two disk solution?

   There are two basic methods: First, you can start the application all over again (re-launch), which is not a bad solution at this point. However, if this were a much larger/longer process you might lose a lot of time re-entering information until you got to this point.

   Another solution is to view and modify variables. This is a quick and easy process.

3. What changed in the Debug (call stack) window?

   The following is a side-by-side illustration of the before and after calls to *do_hanoi()*:

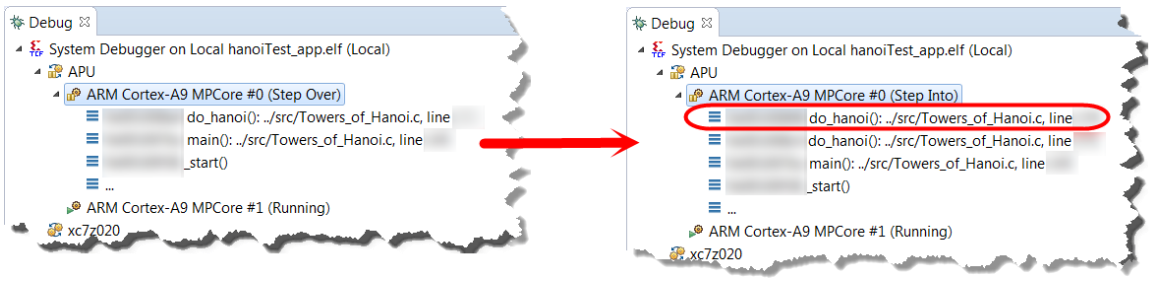


Figure 4-32: Before-and-After Call to do_hanoi()

   Notice that another instance of *do_hanoi()* appears in the call stack. This call stack represents all the different functions that have been pushed onto the stack. If you notice the *_start()* or *_main()* disappearing from this stack, it means that the stack has become corrupted and needs more space, which can be controlled via the linker script.

4. What changed in the Variables tab?

   Any variable that changes value since the last time the debugger was suspended is highlighted in yellow.

5. Should you continue stepping into or stepping over this function?

   Since there is only a small number of disks, you will only need to click a few times before the solution is displayed.

   If you were satisfied with the behavior of the function and there were a large number of disks (and therefore an exponentially large number of clicks) you might want to "step return" from this function. The step return will execute the remainder of the code in the function and stop when the calling function is reached.

6. Do you agree with the solution for the two disk problem?

   Here is what's happening:

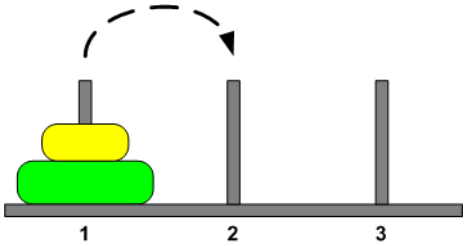


Figure 4-33: Initial Condition and First Move

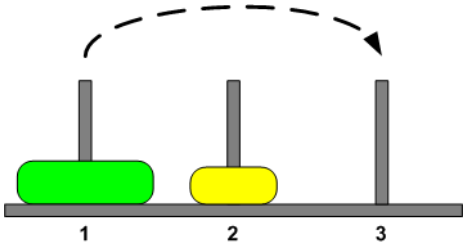


Figure 4-34: Result of First Move and Second Move

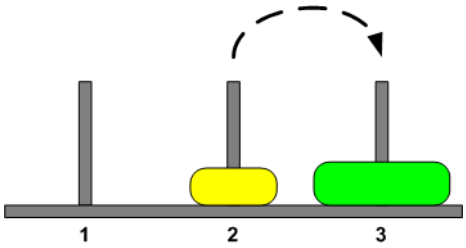


Figure 4-35: Result of Second Move and Third Move

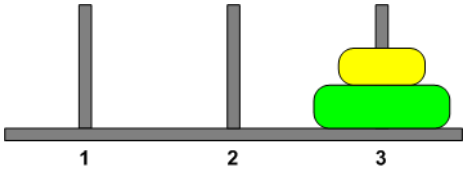


Figure 4-36: Done!

7. What are your choices regarding stopping at a point deep inside a recursive function (or loop)?

   There are three basic approaches:

   - You can single step through the process. This is the least desirable case as it requires a lot of clicking and there is a chance that you might miss the point you want to stop at.
   - You can add a line in your source code that tests for a specific condition and you would have a useless (null) statement inside the true block of the test. You would then set a breakpoint on this useless line of code. The downside to this is that it requires you to add more code, then remove it after the debug session.
   - You can use conditional breakpoints. The only downside is that the evaluation of the expression is done by the debugger and may slow down the execution of the program.

8. How can you tell if a breakpoint is conditional or not?

   There are two basic methods. If the breakpoint is conditional, then:

   - There will be a small question mark next to the breakpoint icon that is visible in both the editor and the Breakpoints tab.
   - In the Breakpoints window, there will be the display of the condition alongside the breakpoint name and line number.

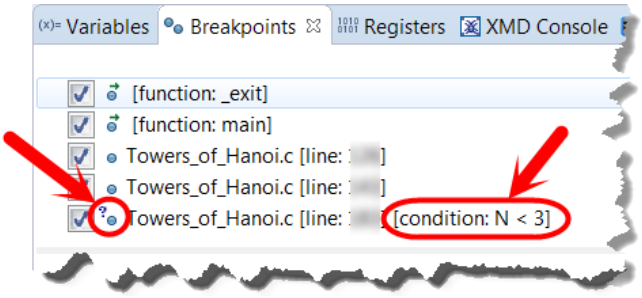


Figure 4-37: Identifying Conditional Breakpoints in the Breakpoint Tab

9. Did the conditional breakpoint work?

   You can verify the value of N in the Variables tab. If the answer is less than three, then the breakpoint worked properly.

10. Was an incursion into the buffer space found?

    Yes, it was found in the 14th word from the base and the value of the incursion was 128 (which is equivalent to 0x80).

Lab 5: Application Development

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Absract

The Xilinx Software Development Kit (SDK) is a powerful tool to help accelerate your software development. This lab demonstrates how to create a simple software application project. The source files for a software loop-based stopwatch are provided and you will fix the code.

Objectives

After completing this lab, you will be able to:

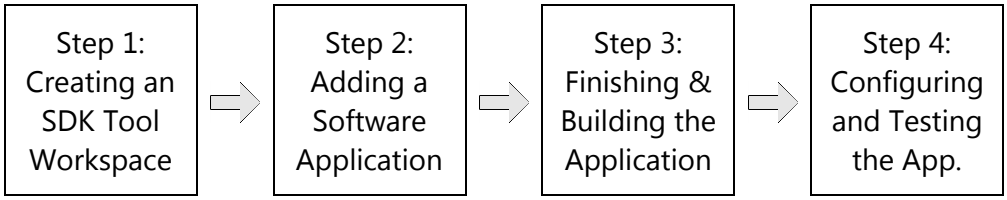
- Create an SDK software application project
- Navigate Xilinx processor peripheral hardware documentation
- Navigate device driver API documentation
- Implement a Level 0 device driver
- Download and run an application on development hardware

Introduction

The software loop-based stopwatch application is mostly written, but it lacks the code to initialize the data direction of two general-purpose I/O peripherals (*axi_gpio)* as either inputs or outputs. One of the five peripherals is for the directional pushbuttons on the FMC-CE I/O board, which you will configure as inputs. The other GPIO drives the LEDs, which you will configure as outputs.

It is your task to find and use the correct Level 0 driver calls that set I/O direction, requiring you to search the hardware and device driver documentation. In this lab, these calls are *#define* macros that have low overhead and execute quickly. The *axi_gpio* is one of the most popular processor peripherals, so much of what you learn here will be useful in future projects.

General Flow



Creating an SDK Tool Workspace Step 1

The first step in this lab is to create an SDK software workspace.

The working directory is where all of the projects (hardware platforms, board support packages, and applications) are stored. This workspace can be placed virtually anywhere. When information is exported from the Vivado® Design Suite, the workspace is created deep within the Vivado Design Suite's hierarchy. You will use a workspace located near the top of the hierarchy for two reasons: first, it is easier to access (fewer clicks), and second, projects are more easily shared.

You will use *C:\training\appDev_standAlone\lab* as the SDK workspace. A hardware project has been built and the exported files are available as a starting point for this lab.

Once SDK is launched and a workspace is set up in this directory, you will not be able to move these directories.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\appDev_standAlone\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\appDev_standAlone\lab** directory and click **OK**.

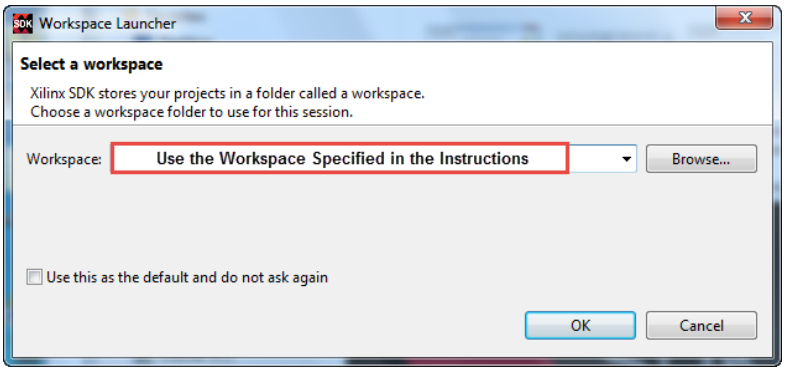


Figure 5-1: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

The hardware platform specification contains a thorough description of the hardware design: what types of processors are present, active peripherals in the PS and PL for Zynq® All Programmable SoC-based systems or a list of all peripherals for a non-Zynq All Programmable SoC system, a full system memory map, etc. Based on this description, software such as the board support package (BSP) and application can be tailored to the hardware.

1-2. Create the hardware platform specification.

1-2-1. Select **File** (1) > **New** (2) > **Other** (3) to see the Xilinx-specific options.

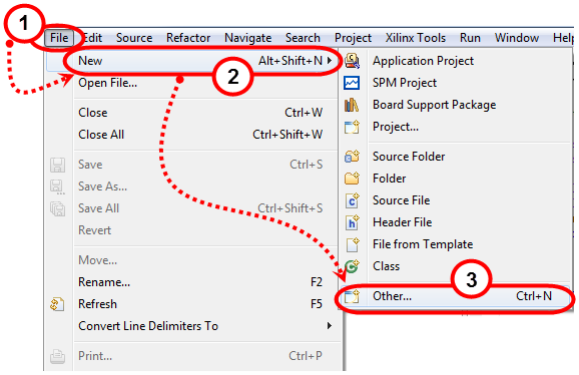


Figure 5-2: Accessing the New Wizards

The Select a Wizard dialog box opens. Here you can select one of many different wizards.

1-2-2. Expand the **Xilinx** folder (1).

1-2-3. Select **Hardware Platform Specification** (2).

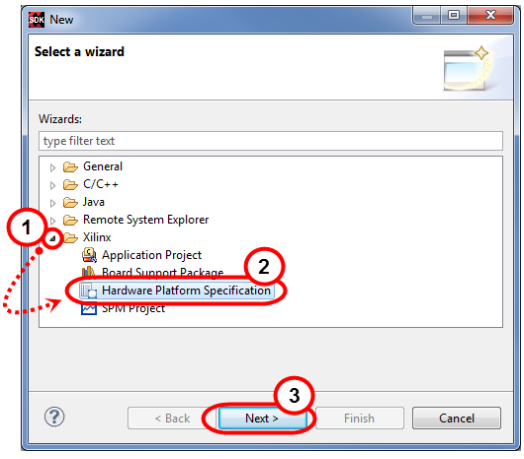


Figure 5-3: Selecting the Hardware Platform Specification Wizard

1-2-4. Click **Next** to open the New Hardware Project dialog box (3).

The New Hardware Project dialog box opens. Here you will be able to specify a project name and the hardware description file that was exported by the Vivado Design Suite.

1-2-5. Enter **UEDfull_hw** in the *Project name* field.

1-2-6. Browse to the *C:\training\appDev_standAlone\support* directory under the Target Hardware Specification region and select the **UEDfull_[zc702 | zed]** file.

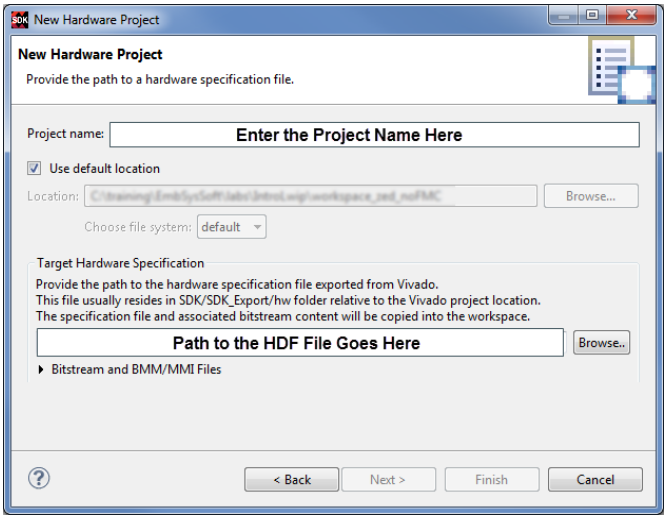


Figure 5-4: New Hardware Project Dialog Box

1-2-7. Click **Finish** to create the new hardware project.

Adding a Software Application — Step 2

Now you will generate the software platform for the hardware and an empty software project. Then you will import existing C source files into the project and SDK will automatically build and produce an Executable and Load Format (ELF) file.

You will examine, but not change, the board support package (BSP) platform options. Nearly complete C source code and associated LCD drivers are provided which you will add to the project. Then you will generate a linker script and configure the compiler options.

As a first step in a new software environment, a software platform must be built to contain a library of system and processor services, as well as device drivers for the hardware peripherals that make up the system. Once a software platform has been built, then a software application can be written. Xilinx provides the Software Platform Wizard to accomplish this task.

Using the Application Project Wizard is a quick way to set up a C or C++ software application project that targets an existing processor and OS platform (Standalone or Linux). You can automatically generate the board support package

(BSP) or select an existing one. Based on the dialog box choices, the appropriate tool chain is selected for pre-processing, compiling, assembling, and linking.

2-1. Create a new C/C++ application project named *appDev_app*. Use the board support package named *UEDfull_bsp*.

2-1-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

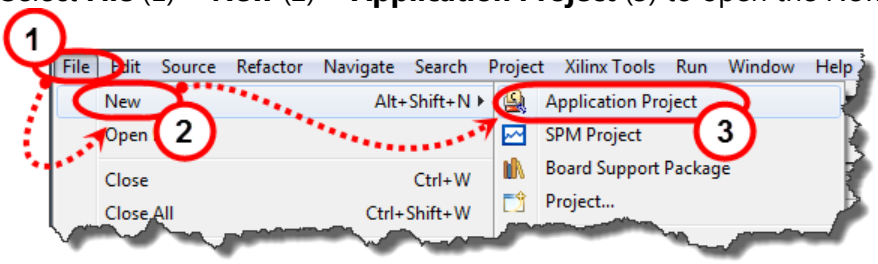


Figure 5-5: Creating an Application Project

2-1-2. Enter **appDev_app** as the project name.

2-1-3. Ensure that you have **UEDfull_hw** selected from the Hardware Platform drop-down list as the SDK tool can manage multiple platforms within a single workspace.

This will populate the Processor drop-down list accordingly.

2-1-4. Ensure that **ps7_cortexa9_0** is selected from the Processor drop-down list.

2-1-5. Ensure that **standalone** is selected from the OS Platform drop-down list.

2-1-6. Select **Use Existing** and choose an existing BSP from the drop-down list if you have already created a BSP for this hardware platform; otherwise, select **Create New** and enter the name for the BSP as **UEDfull_bsp**.

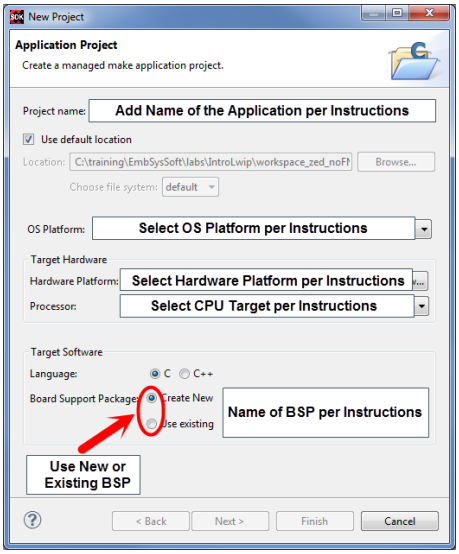
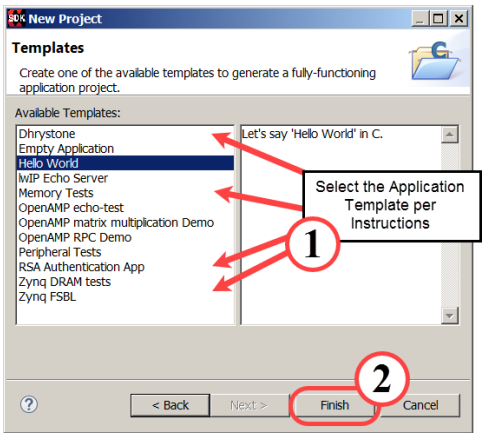


Figure 5-6: Entering Application Project Information

2-1-7. Click **Next** to select the template for this application.

2-1-8. Select **Empty Application** (1).



Figureb 5-7: Selecting an Application Template (Selection in Figure May Not Match Instructions)

2-1-9. Click **Finish** to create the new project (2).

As the project is created, the new BSP is compiled and any sources from the templates are also compiled. The creation of the new project usually takes less than a minute.

It will take a minute or two to compile the board support package (BSP). Should you ever need to add libraries to the BSP, you can right-click the BSP project and select **Board Support Package Settings**. From there you will be able to make changes to the STDIO mappings, add and change libraries and settings within those libraries, etc.

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

2-2. Add *stopwatch_app.c, lcd.c, lcd.h* to the application.

The preferred method for importing sources is shown here.

2-2-1. Expand the project named **appDev_app** > **src** using the Project Explorer.

2-2-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

2-2-3. Select **Import** to open the Import Wizard (2).

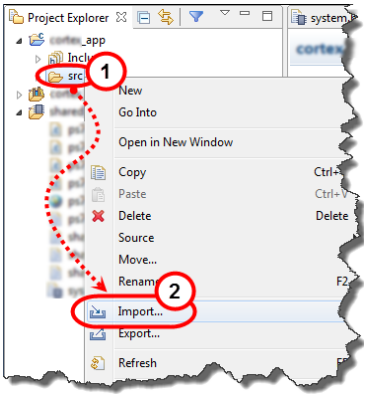


Figure 5-8: Importing a Resource File

The Import Wizard dialog box opens.

2-2-4. Expand **General** (1).

2-2-5. Select **File System** as you will be selecting individual files directly from the file system (2).

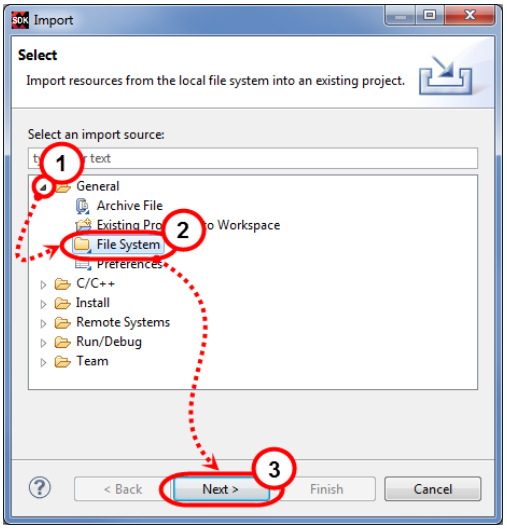


Figure 5-9: Selecting File System

2-2-6. Click **Next** to advance to specifying the files to import (3).

2-2-7. Browse to *C:\training\appDev_standAlone\support* in the *From directory* field.

2-2-8. Select the file(s) by checking the box beside **stopwatch_app.c, lcd.c, lcd.h**.

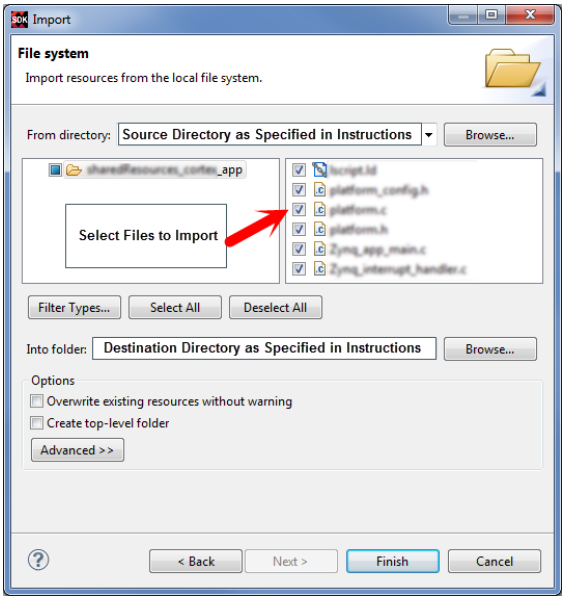


Figure 5-10: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

2-2-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

2-3. Verify that the application builds correctly.

2-3-1. Select the **Console** tab and ensure that the application is built without errors.

If there are any errors, they will be noted here.

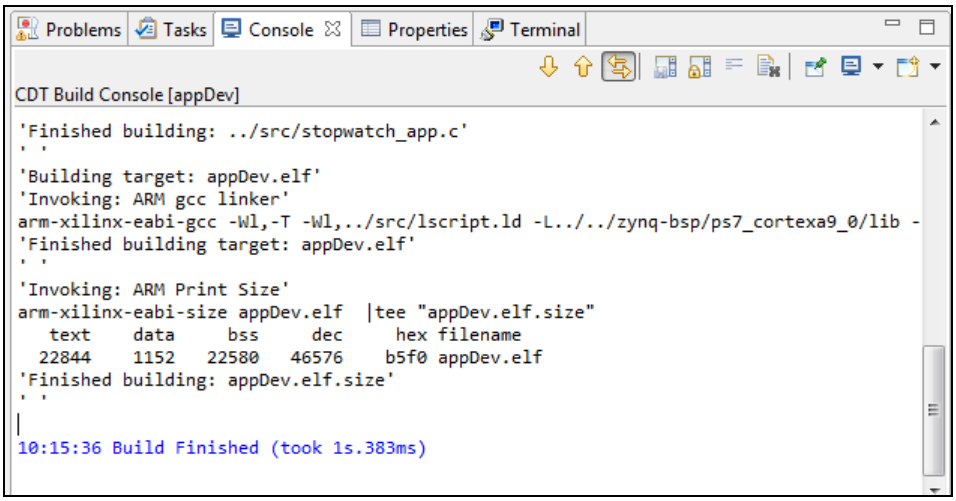


Figure 5-11: Successful Software Application Build

Question 1

When might you change a setting in the Board Support Package Settings dialog box and regenerate the BSP?

Question 2

What would you do differently in this step to generate faster, more compact C code?

Finishing and Building the Software Application Step 3

The stopwatch application is almost complete. Only the configuration of the registers controlling the data direction of the *axi_gpio* peripherals used for the pushbutton inputs and LED outputs need to be initialized.

3-1. Review the *axi_gpio* registers to determine how to program I/O data direction.

3-1-1. Expand the **UEDfull_hw** hardware platform specification project in the Project Explorer tab.

3-1-2. Double-click **system.hdf** to open the hardware platform specification in the edit window.

3-1-3. Scroll down to the **IP blocks present in the design** section.

3-1-4. Continue to scroll down this list in order to find the GPIO_Buttons_rosetta IP.

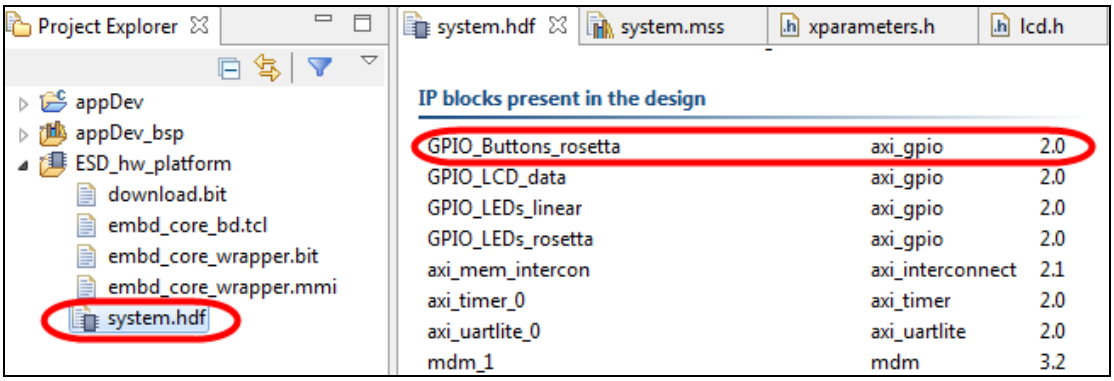


Figure 5-12: Viewing the IP Blocks Present in the Design [MicroBlaze Processor]

To the right of *GPIO_buttons_rosetta* (or any peripheral that uses the gpio peripheral) is a version number about the selected IP and, depending on the IP, there may be a registers hyperlink.

3-1-5. Click the **registers** hyperlink to see the available registers and their offsets.

While this table provides some information, you may want additional information about this piece of IP.

3-2. Access the documentation on the AXI_GPIO peripheral.

The documentation provided in this section is the detailed discussion of the hardware and software drivers. This location will provide the most detailed information on the IP that you select.

3-2-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **DocNav** > **DocNav**.

3-2-2. Select **IP** from under IP.

3-2-3. Expand **Document Types** and select **Product Guides**.

3-2-4. Expand **Functional Categories** and select **AXI**.

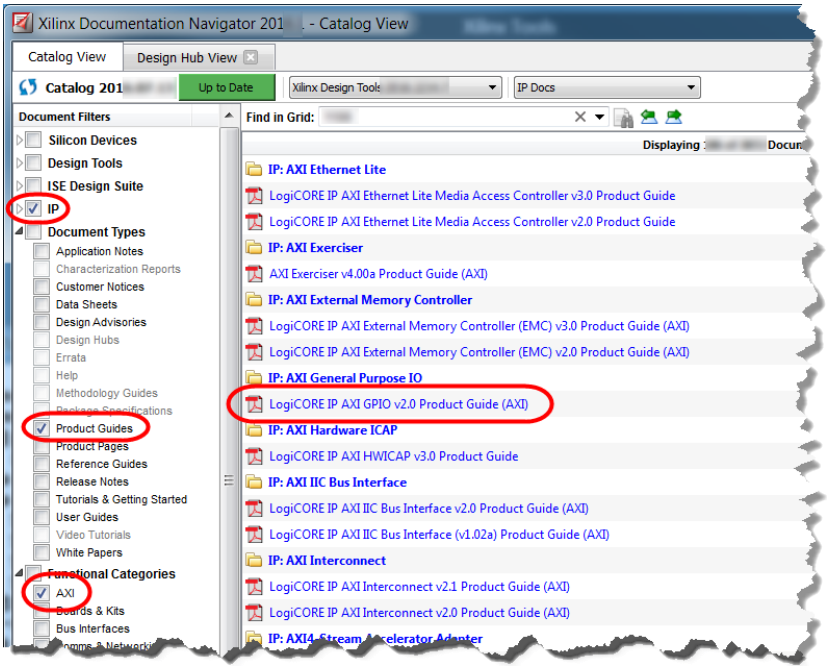


Figure 5-13: Selecting the IP Documents in DocNav

3-2-5. Click the **LogiCORE IP AXI GPIO (v2.0) Product Guide** link to open the GPIO datasheet in a new tab in DocNav.

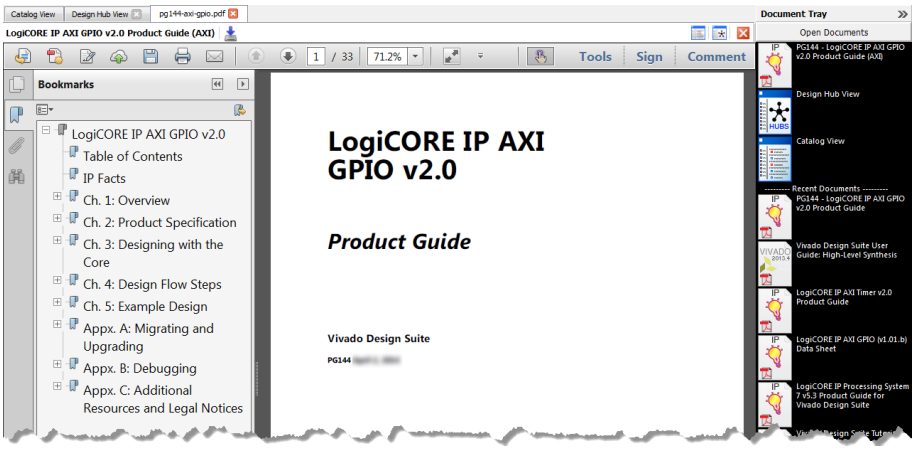
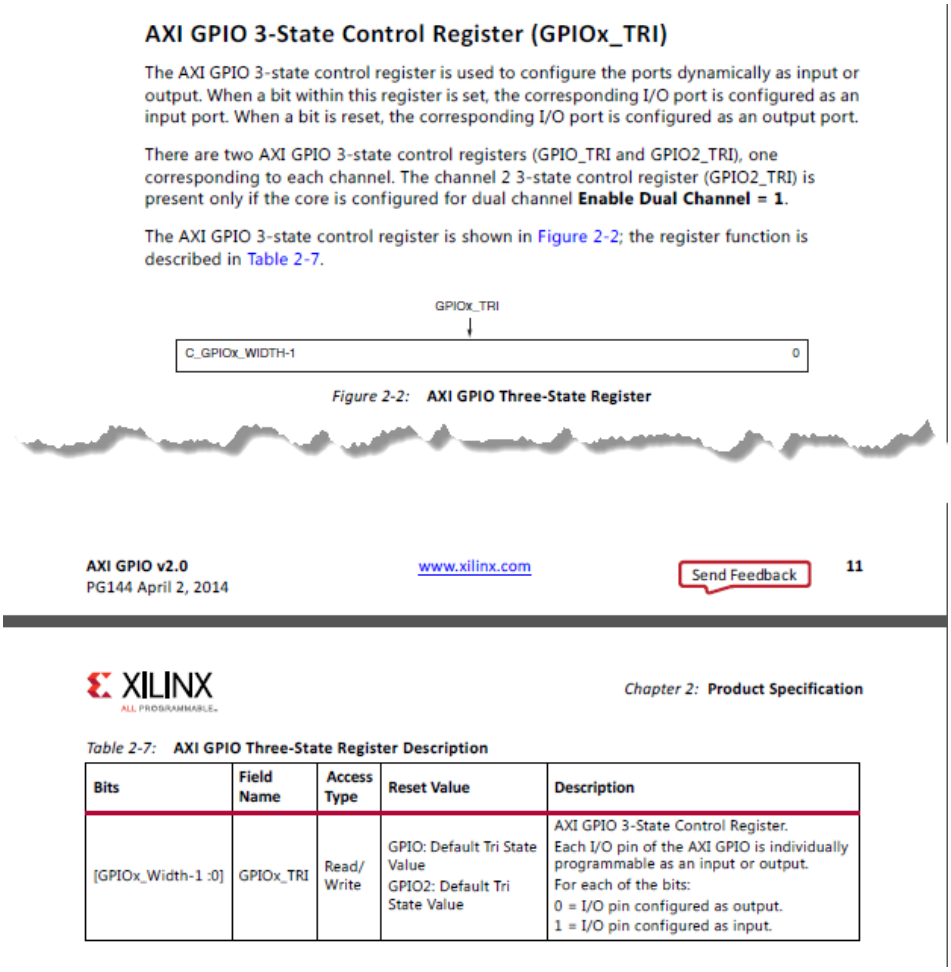


Figure 5-14: axi_gpio Datasheet

3-2-6. View the operation of the AXI GPIO 3-State Register under the "AXI GPIO 3-State Control Register (GPIOx_TRI)" topic in the product guide.



AXI GPIO 3-State Control Register (GPIOx_TRI)

The AXI GPIO 3-state control register is used to configure the ports dynamically as input or output. When a bit within this register is set, the corresponding I/O port is configured as an input port. When a bit is reset, the corresponding I/O port is configured as an output port.

There are two AXI GPIO 3-state control registers (GPIO_TRI and GPIO2_TRI), one corresponding to each channel. The channel 2 3-state control register (GPIO2_TRI) is present only if the core is configured for dual channel **Enable Dual Channel = 1**.

The AXI GPIO 3-state control register is shown in Figure 2-2; the register function is described in Table 2-7.

Figure 2-2: **AXI GPIO Three-State Register**

Table 2-7: **AXI GPIO Three-State Register Description**

Bits	Field Name	Access Type	Reset Value	Description
[GPIOx_Width-1 :0]	GPIOx_TRI	Read/ Write	GPIO: Default Tri State Value GPIO2: Default Tri State Value	AXI GPIO 3-State Control Register. Each I/O pin of the AXI GPIO is individually programmable as an input or output. For each of the bits: 0 = I/O pin configured as output. 1 = I/O pin configured as input.

Figure 5-15: 3-State Register Information for the AXI GPIO

It states that bits in the register cleared to '0' configure the associated I/O pin as an output and bits set to '1' configure the associated I/O pin as an input.

Now you need a way to set up the registers in the *stopwatch_app.c* program. For the sake of simplicity, you will use a low-level macro.

3-3. Often, from the programmer's perspective, the details of the hardware or registers may not be needed, rather, a listing of the APIs and how they are used may be of more value.

The following instructions will guide you through accessing the peripheral documentation and locating the necessary drivers.

3-3-1. Select the SDK icon ( ) from the Taskbar to make it active if the SDK is not the active window.

3-3-2. Expand the **UEDfull_bsp** project to expose the *system.mss* file.

3-3-3. Double-click **system.mss** to open the board support package information in the edit window.

3-3-4. Scroll down to the Peripheral Drivers section.

3-3-5. Locate any of the GPIO peripherals (while they have different purposes within the design, they all use the same IP and hence, have the same documentation).

3-3-6. Click the **Documentation** link next to a gpio peripheral from the list of peripheral drivers to access the driver documentation in a browser window.

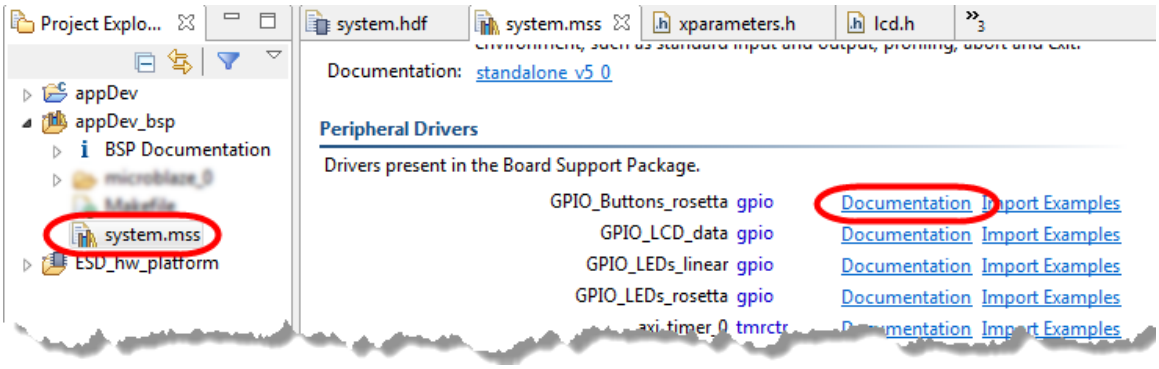


Figure 5-16: system.mss File – Board Support Package Information

3-3-7. Select the **APIs** tab to see all of the available APIs for the device.

3-3-8. Locate an API with a name that suggests that it will write to the registers, since you will need to write to the registers within the GPIO peripheral to control it (low level).

3-3-9. Click the hyperlink next to **XGpio_WriteReg** to see the documentation for that specific API call.

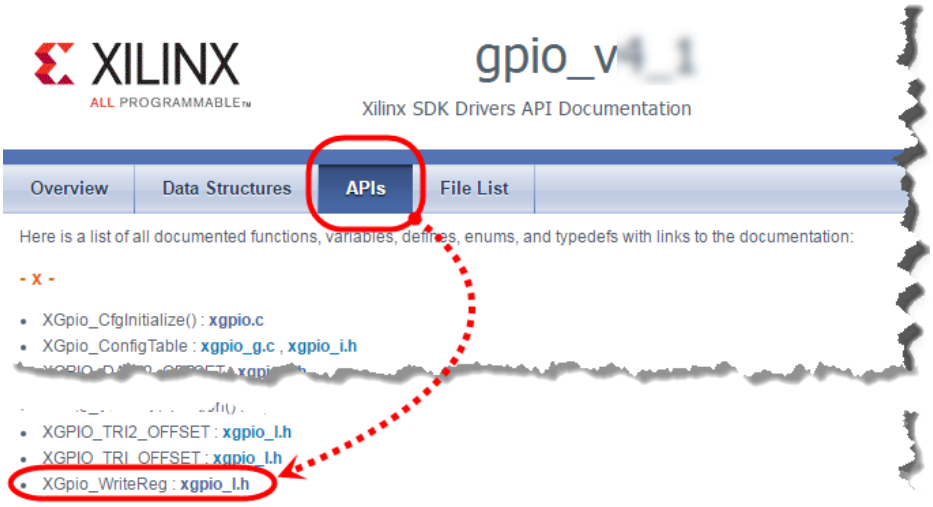


Figure 5-17: Locating the XGpio_WriteReg API

The *xgpio_l.h* file reference document opens in the browser window, displaying information about the low-level write register macro.

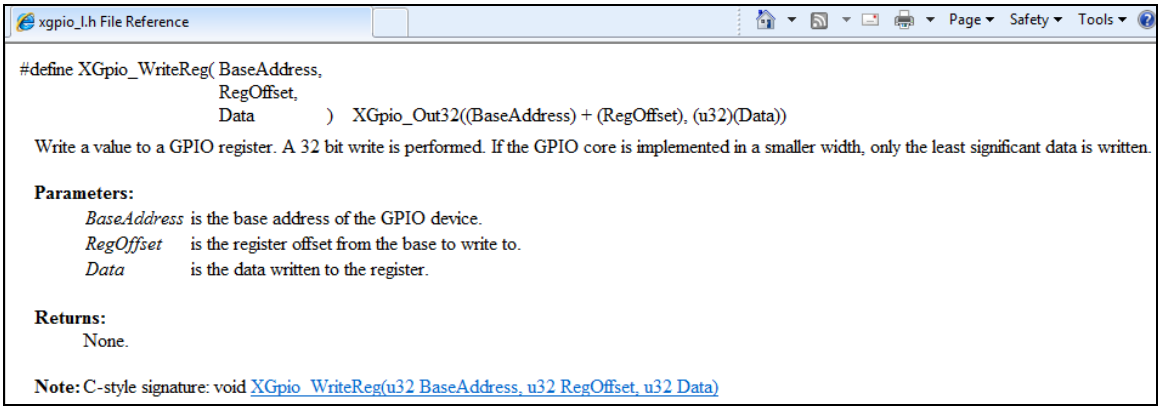


Figure 5-18: Low-Level Write Register Macro Description

Note that all the Level 0 macros are contained in *xgpio_l.h*.

3-3-10. Click the **xgpio_l.h** hyperlink next to the #include to access the **xgpio_l.h** file.

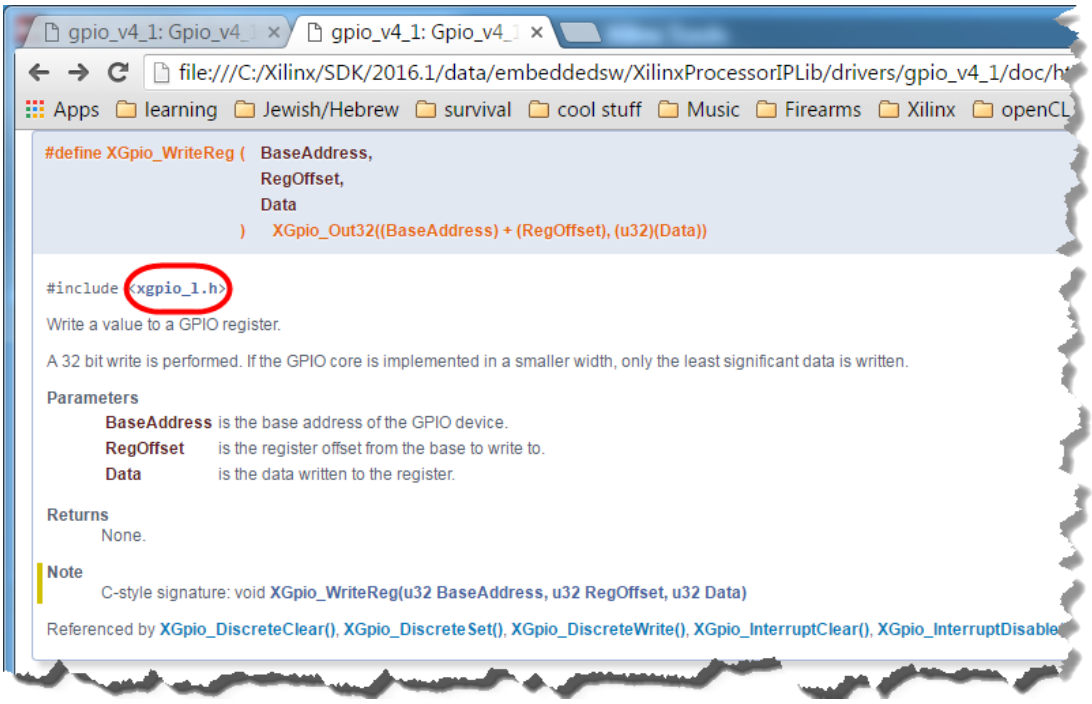


Figure 5-19: Accessing the Header File Documentation

3-3-11. Scroll through the *xgpio_l.h* file reference information.

Note the register offset *#define* statements that are useful when you are using the low-level macros.

3-3-12. Study the macro and the *#define* symbols that need to be used to set the gpio data direction.

You will now complete the provided application code that can be found in the Project Explorer tab under appDev_app > src.

3-4. Make the SDK window active.

3-4-1. Select the SDK icon (

) from the taskbar to make it active if the SDK is not the active window.

3-5. Open *stopwatch_app.c* in the editor.

3-5-1. Locate **stopwatch_app.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

3-5-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

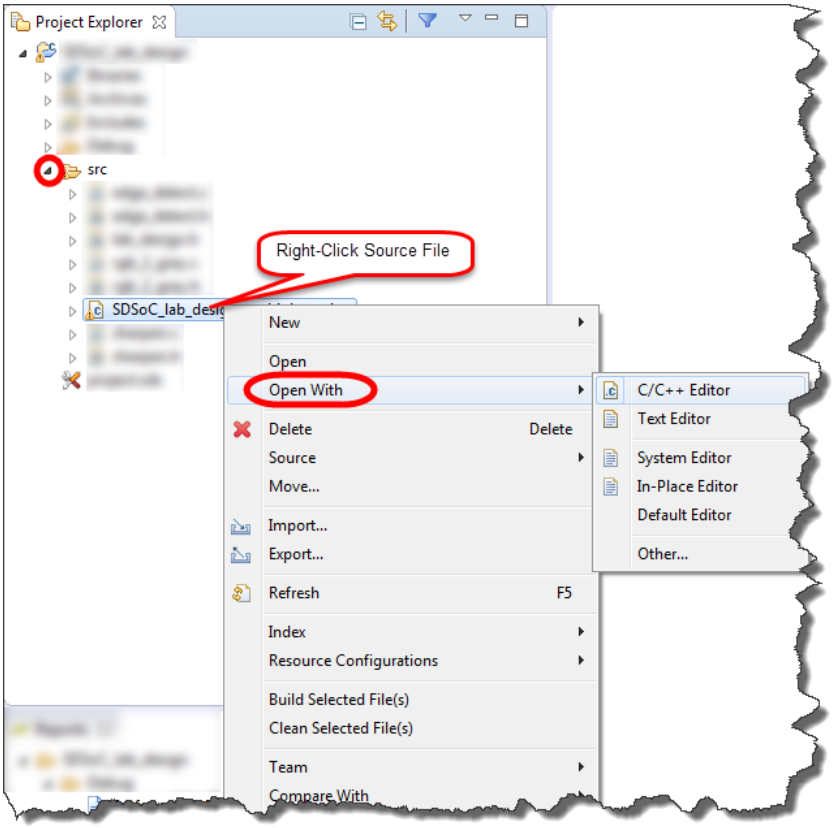


Figure 5-20: Opening a Source File via Right-Click

3-5-3. Right-click in the left margin of the *stopwatch_app.c* file to open the context menu.

3-5-4. Select **Show Line Numbers** to enable line numbering.

3-6. Find "*Student to add code here*".

The find/replace operation is accessible through both a click sequence and a keyboard shortcut.

3-6-1. Select **Edit** > **Find/Replace** or press <**Ctrl** + **F**>.

The Find/Replace dialog box opens.

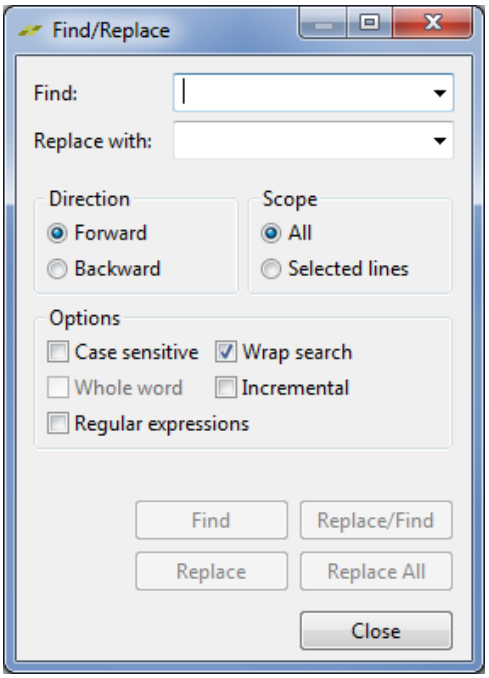


Figure 5-21: Default Find/Replace Dialog Box

3-6-2. Enter **"Student to add code here"** in the Find field.

3-6-3. Click **Find** or press <**Enter**> to find the next occurrence of "*Student to add code here*".

3-6-4. Continue clicking **Find** or pressing <**Enter**> until you locate the specific instance that you are looking for.

With the **Wrap search** option enabled, the file is treated as a continuous loop and the find operation will jump to the next occurrence at the top of the file (when searching forwards) or at the bottom of the file (when searching backwards). A "ding" sound is made when the search wraps around.

If you are looking for text within a specific region of the code you would first highlight the region to perform the search in, then launch the Find/Replace function as described in this topic.

3-6-5. Click **Close** to close the Find/Replace dialog box.

3-7. Write the API (macro driver functions/low level) call to set the *axi_gpio* data direction for the pushbuttons and LEDs.

You will need the base address of the 8-bit LEDs and 5-bit pushbutton peripherals. The base address #defines are found in the *xparameters.h* file that is automatically created when the BSP is built.

3-7-1. Open **lcd.h** as described above.

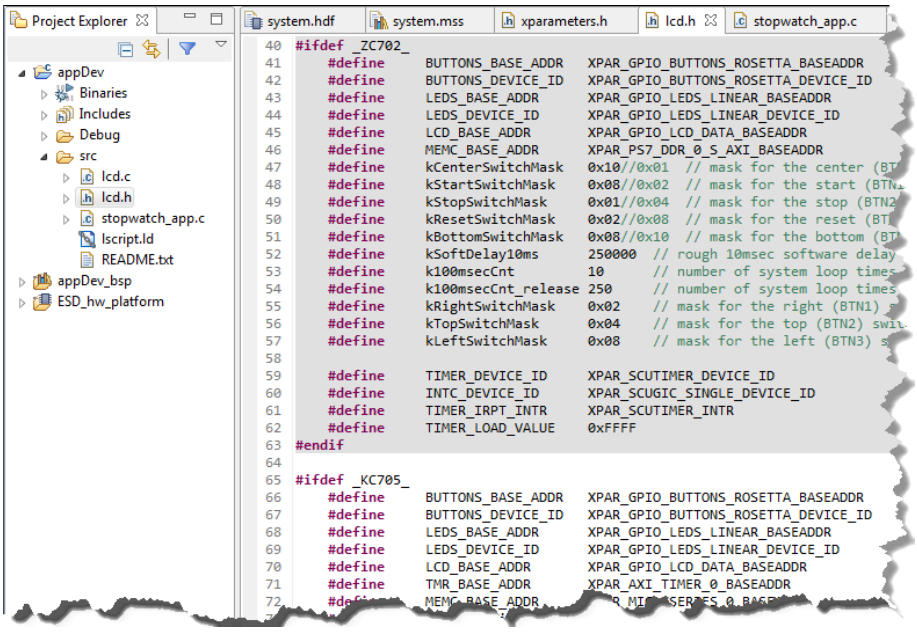


Figure 5-22: Conditionally Compiled Blocks of Code

Note that conditionally compiled blocks of code that were not compiled are shown with a gray background so that you can easily tell what code was compiled.

3-8. Write the code to set the direction for the pushbutton and LED.

3-8-1. Return to the *stopwatch_app.c* tab where you just found the comment blocks that begin with "*Student to add code here*".

3-8-2. Add the macro calls necessary to initialize the data direction registers of the pushbutton and LED gpio peripherals.

You can also copy and paste the necessary code from the *axi_gpio_macro_calls.c* file in the *C:\training\appDev_standAlone\support* directory.

```
//*************************************
// Set the direction for all Push Button signals to be inputs
XGpio_WriteReg(BUTTONS_BASE_ADDR, XGPIO_TRI_OFFSET, 0xffffffff);

// Set the direction for all LED signals to be outputs
XGpio_WriteReg(LEDS_BASE_ADDR, XGPIO_TRI_OFFSET, 0);
```

Figure 5-23: Macro Calls to Initialize the Data Direction Registers

3-8-3. Save the **stopwatch_app.c** file when you finish.

The tools will automatically rebuild the project.

A successful build is indicated when the program size is returned.

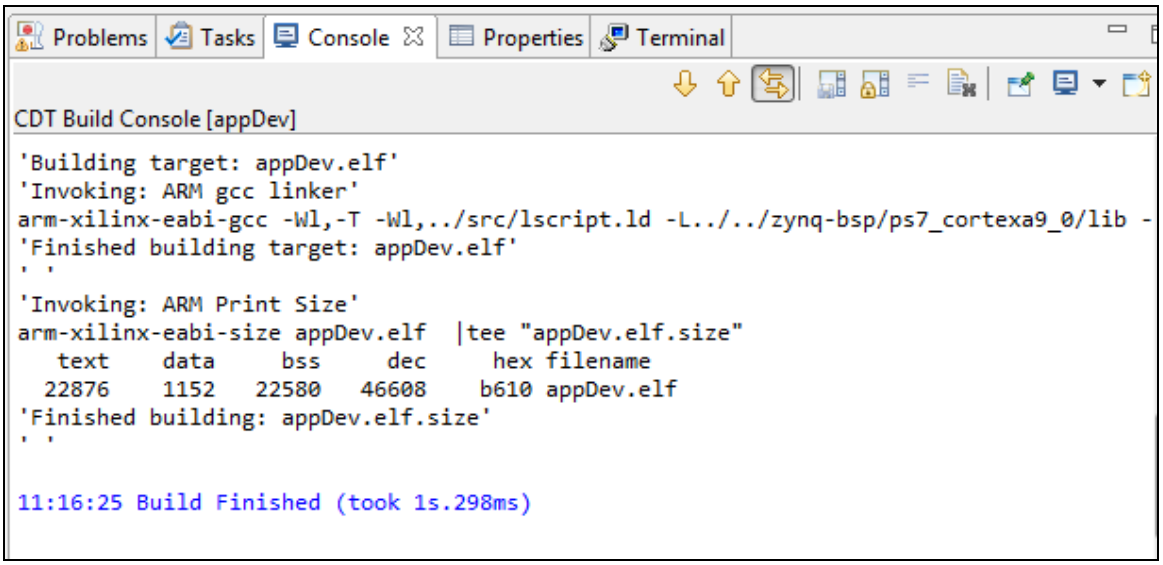


Figure 5-24: Returning the Program Size Indicates a Successful Build (Zynq AP SoC)

Note that your code size may vary due to the specific version of the tools.

Question 3

In what file are the low-level macro-based gpio functions found? How many gpio macro functions exist?

Question 4

Is changing, editing, or adding to the *xparameters.h* file a good idea? Why or why not?

Configuring the Device and Testing the Application Step 4

Now that the finishing touches have been put on the application code, you will configure the Zynq AP SoC programmable logic (PL) or FPGA device, download the application, and test the program for proper stopwatch operation. The hardware portion of the design has already been implemented for you. Ensure that the hardware board is on and both the programming and serial communication cables are properly connected.

4-1. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

4-1-1. Select **Xilinx Tools** > **Program FPGA** or click the  icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

4-1-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

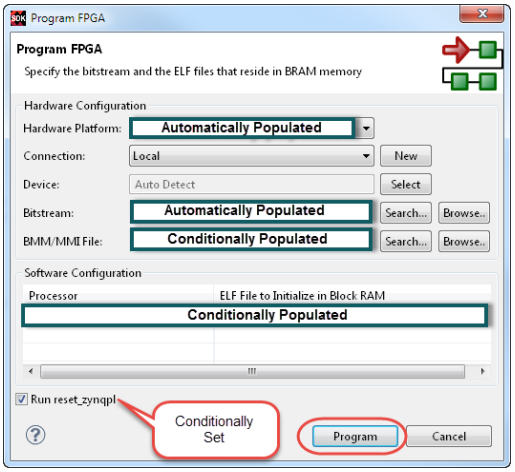


Figure 5-25: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

4-2. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

4-2-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

4-3. Configure the SDK Terminal.

4-3-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

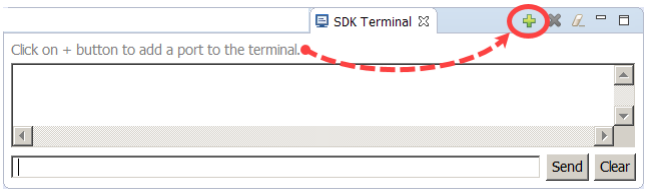


Figure 5-26: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

4-3-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

4-3-3. Set the baud rate to **115200** (2).

4-3-4. Leave the other settings at their defaults.

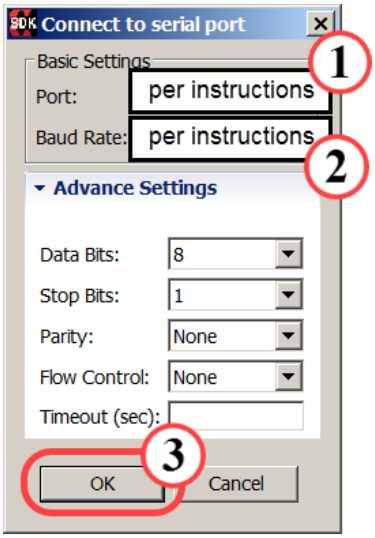


Figure 5-27: Configuring the SDK Terminal

4-3-5. Click **OK** to save these settings and begin the terminal session (3).

Configurations are a very powerful ally in setting up a run for a very specific set of criteria. Often, however, the default configuration settings are adequate and there is no need to click through a number of dialogs just to run the program.

4-4. Run *appDev_app* with the default configuration settings.

4-4-1. Right-click **appDev_app** in the Project Explorer to open the context menu.

4-4-2. Select **Run As** > **Launch on Hardware (System Debugger)** to immediately launch the application on the hardware.

Note: Selecting **Run As** > **Launch on Hardware (GDB)** will also work; however, this is a deprecated flow.

If an application is already running, a warning message will appear asking you to terminate the previous session. If you receive this message, click **Yes** to terminate the running application and run the new scenario.

The application runs.

--- Alternate Method ---

Select **appDev_app** in the Project Explorer tab.

Once the application project is selected, you can launch the run by:

- Use the menu bar to select **Run** > **Run**.
  - Then select one of the options from the **Run As** window (this window will pop up only if there is no Run configuration).
- Press <**Ctrl** + **F11**>.
- Click the **Run** icon ( ).

The stopwatch time appears on the serial output. The eight LEDs should increment left to right when the stopwatch is running.

4-5. Verify the program.

4-5-1. Verify proper stopwatch operation.

The buttons are located on the FMC-CE card and on the board. Their behavior is defined as:

- START → BTN1 (Left)
- STOP → BTN2 (Select)
- RESET → BTN3 (Right)

4-6. Terminate the program.

4-6-1. Click the **Debug** perspective button.

Note: If it is not visible, select **Window** > **Open Perspective** > **Debug**.

4-6-2. Click the **Disconnect** ( ) button.

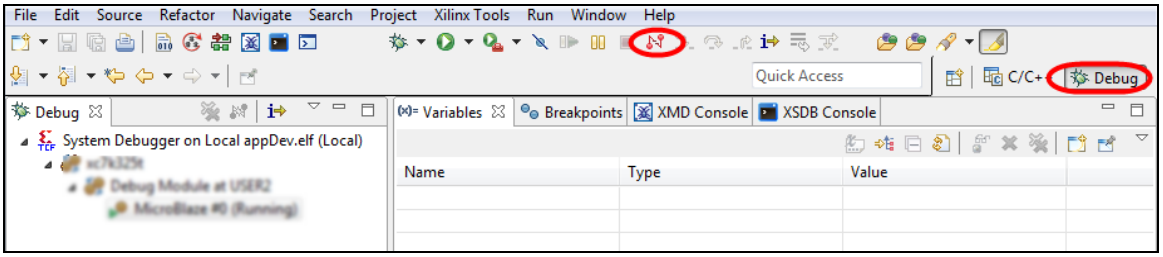


Figure 5-28: Disconnecting the Application

4-7. Exit the SDK tool and power off the board.

4-7-1. Select **File** > **Exit** to close the SDK tool.

4-7-2. Power off the development board.

Question 5

The Run configuration is set for the Debug project configuration. What is the purpose for all of these configurations?

Summary

You created a basic SDK software application that operates a software loop-based stopwatch. The application utilized the Level 0 *axi_gpio* device drivers. You used the documentation available from SDK to investigate how to identify and install these drivers in the C source. You also learned how to configure the FPGA, then load and run the software application.

Answers

1. When might you change a setting in the Board Support Package Settings dialog box and regenerate the BSP?

   The settings can be changed for various reasons, including:

   - Changing hardware
   - Changing the configuration options of the standalone BSP services
   - Changing or adding a new device driver to the BSP

2. What would you do differently in this step to generate faster, more compact C code?

   The properties in the C/C++ Build section had the compiler optimization turned off for maintaining coherency between the source and the executable. Setting the optimization level higher will cause the compiler to generate more efficient and compact code, but it is difficult to single step.

3. In what file are the low-level macro-based gpio functions found? How many gpio macro functions exist?

   Low-level, macro-based functions for the gpio peripheral are all found in the *xgpio_l.h* file.

   There are only two gpio macro functions: *XGpio_WriteReg( )* and *XGpio_ReadReg( )*. These macros are useful for doing simple tasks in small programs. In larger, more complex programs, the gpio functions found in the *xgpio.c* file may be a better choice.

4. Is changing, editing, or adding to the *xparameters.h* file a good idea? Why or why not?

   No, the *xparameters.h* file is regenerated every time the BSP is built by LibGen. This is a read-only file.

5. The Run configuration is set for the Debug project configuration. What is the purpose for all of these configurations?

   SDK accommodates multiple setups for quickly choosing an object ELF application to download to the simulator or hardware; this is a RUN configuration. Project configurations, such as Release, Debug, and Profile, each allow the compiler/linker to generate a different ELF file based on tool settings for each configuration. The Debug configuration is created by default during project creation.

Lab 6: File Systems

Zynq All Programmable SoC ZC702 or ZedBoard or Kintex-7 FPGA KC705

2016.3

Abstract

The goal of this lab is to upgrade an embedded system design to have the capability of performing memory file system functions in an external DDR memory. This lab guides you through the principles of creating a file system using the XilMFS library. You will create a software platform utilizing the provided hardware design. Using SDK, you will also analyze and implement an application that creates files and directories in DDR memory.

Objectives

After completing this lab, you will be able to:

- Create a software platform that makes use of the Xilinx Memory File System XilMFS
- Use some of the XilMFS functions to perform file system operations
- Demonstrate a working, file system-based embedded system

Introduction

A memory file system (MFS) provides the capability to manage program memory. The file system includes directories and files within each directory. It can be accessed from the high-level C language through functional calls specific to the file system. Alternatively, files can be managed through the standard C language functions provided in XilFile.

The goal of this lab is to upgrade an embedded system design with memory file system functions capabilities in an external DDR memory. You will use the evaluation board and the Serial Port to demonstrate file system operations.

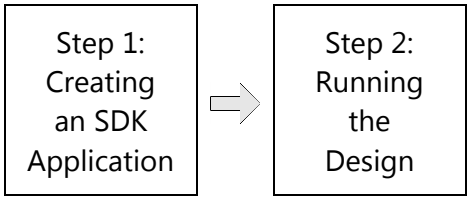
General operation is as follows:

1. Displays a welcome message on the serial port when the embedded application starts.
2. Waits for serial port commands to be delivered.
3. Depending on the serial command entered, does one of the following:
   - 'C': Creation of a directory called *testdir1*, if one is not already created. If the directory already exists, displays an appropriate message.
   - 'D': Deletes a directory called *testdir1*, if it exists. Otherwise, displays an appropriate message.
   - 'c': Creates a file called *testfile1*, writes a test message into it, and closes the file. If the file already exists, displays an appropriate message.

- 'r': Reads a file called *testfile1* and displays the contents. If the file does not exist, displays an appropriate message.
- 'd': Deletes a file called *testfile1* if it exists. If the file does not exist, displays the appropriate message.

All messages should display in the serial port monitor.

General Flow



Creating an SDK Application that Uses the Memory File System Step 1

There are three basic software tasks that will need to be performed: BSP customization, linker script customization, and application development.

Before proceeding, confirm that the hardware board and download cabling are in place.

Several common and repetitive tasks have been automated with a custom tool called *SDK Launcher*. This tool will automatically create an SDK workspace, create a hardware platform specification targeting the desired development board and processor combination and, finally, launch the SDK tool in the open workspace.

The hardware platform specification will be named *UED_hw*. If you are unsure how to do any of these tasks manually, reference the relevant sections in the *Lab Reference Guide*.

1-1. Open the SDK Launcher tool.

1-1-1. Open Windows Explorer and browse to the *C:\training\tools* directory.

1-1-2. Double-click the **SDKlauncher.jar** file to launch the application.

1-2. Configure the SDK Launcher.

1-2-1. Click the Browse icon button to open a file browser window (1).

1-2-2. Select *C:\training\MemoryFileSystem* to identify which lab to work with.

1-2-3. Click **Open** to select the path and return to the tool's main GUI.

1-2-4. Click the Down arrow next to the platform list to expand the list (2).

1-2-5. Select the **PS | MicroBlaze on ZC702 | ZedBoard.**

Note: Selecting PS will target the ARM® processor and MicroBlaze will target the MicroBlaze™ processor.

1-2-6. Ensure that Vivado version **2016.3** is populated in the SDK Version field (3).

This instructs the SDK Launcher application to use a particular version of the SDK tool.

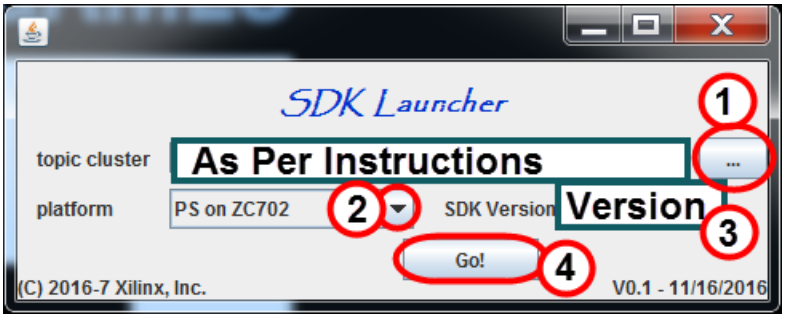


Figure 6-1: Configuring the SDK Launcher Tool

1-2-7. Click **Go** to automatically set up the workspace, create the hardware platform specification, and launch SDK (4).

1-2-8. Close the Welcome screen once SDK launches, if it is open.

Using the Application Project Wizard is a quick way to set up a C or C++ software application project that targets an existing processor and OS platform (Standalone or Linux). You can automatically generate the board support package (BSP) or select an existing one. Based on the dialog box choices, the appropriate tool chain is selected for pre-processing, compiling, assembling, and linking.

1-3. Create a new C/C++ application project named *fileSys_app*. Use the board support package named *fileSys_bsp*.

1-3-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

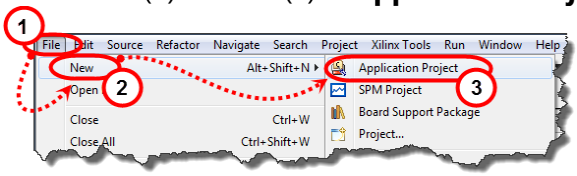


Figure 6-2: Creating an Application Project

1-3-2. Enter **fileSys_app** as the project name.

1-3-3. Ensure that you have **UED_hw** selected from the Hardware Platform drop-down list as the SDK tool can manage multiple platforms within a single workspace.

This will populate the Processor drop-down list accordingly.

1-3-4. Ensure that **ps7_cortexa9_0 for Zynq (MicroBlaze will be supported in 2017.1)** is selected from the Processor drop-down list.

1-3-5. Ensure that **standalone** is selected from the OS Platform drop-down list.

1-3-6. Select **Use Existing** and choose an existing BSP from the drop-down list if you have already created a BSP for this hardware platform; otherwise, select **Create New** and enter the name for the BSP as **fileSys_bsp**.

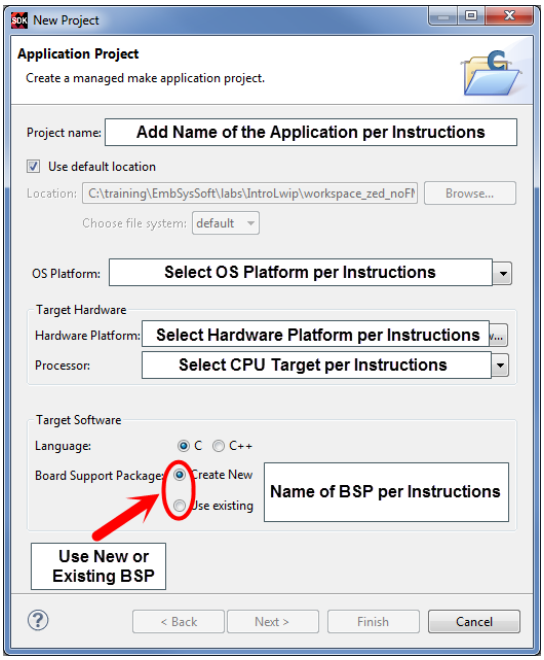


Figure 6-3: Entering Application Project Information

1-3-7. Click **Next** to select the template for this application.

1-3-8. Select **Empty Application** (1).

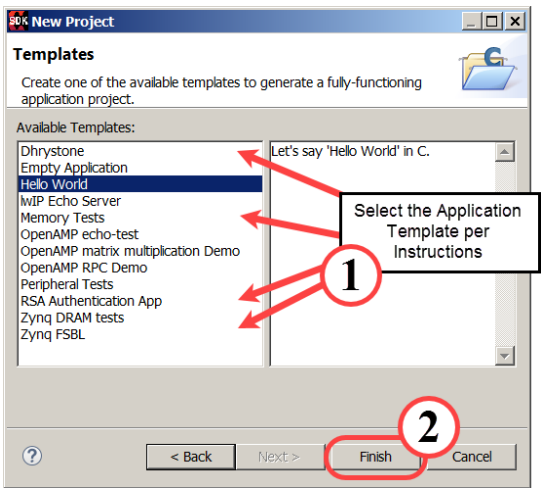


Figure 6-4: Selecting an Application Template (Selection in Figure May Not Match Instructions)

1-3-9. Click **Finish** to create the new project (2).

As the project is created, the new BSP is compiled and any sources from the templates are also compiled. The creation of the new project usually takes less than a minute.

Make sure that you had created a new application by selecting the **Create New BSP** option because you will have to customize the new board support package (BSP).

1-4. Set the tools to build the project(s) any time they are updated.

Typically, the Auto-Build feature is turned on; however, since the SDKlauncher tool ran a Tcl script that built the starting point for this lab, the Auto-Build feature is turned off.

1-4-1. Click **Project** in the menu bar to access the options available for project building.

1-4-2. Select **Build Automatically** to enable the auto-build function.

1-5. Configure the BSP to use the Xilinx Memory File System (xilmfs) and set the number of bytes used by the file system. This represents the number of bytes of memory available for the file system. You can use any size based on the available DDR size. For this lab, use an arbitrarily selected value of 64 MB for the file system which is far more than enough for the few files and will fit in the available DDR memory space.

1-5-1. Right-click **fileSys_bsp** in the Project Explorer window.

1-5-2. Select **Board Support Package Settings**.

1-5-3. Select **xilmfs** (for the Xilinx Memory File System) from the Overview view tab under Supported Libraries.

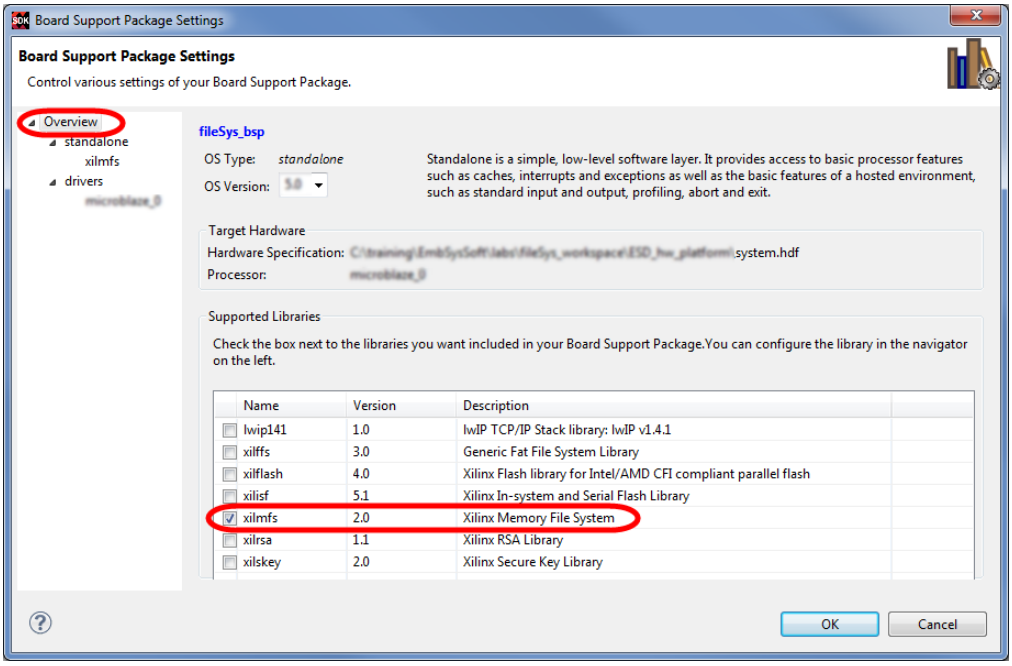


Figure 6-5: Selecting the Xilinx Memory File System

1-5-4. Expand **Overview** > **standalone** in the left hand margin.

1-5-5. Select **xilmfs** to configure the file system library.

1-5-6. Set the value of the *numbytes* parameter to **67,108,864** (64 MB).

This value will be updated in the BSP driver file *mfs_config.h* as #define MFS_NUMBYTES <value>.

Zynq AP SoC users: This file located in the *fileSys_bsp* > *ps7_cortexa9_0* > *include* directory.

MicroBlaze™ processor users: This file located in the *fileSys_bsp* > *microblaze_0* > *include* directory.

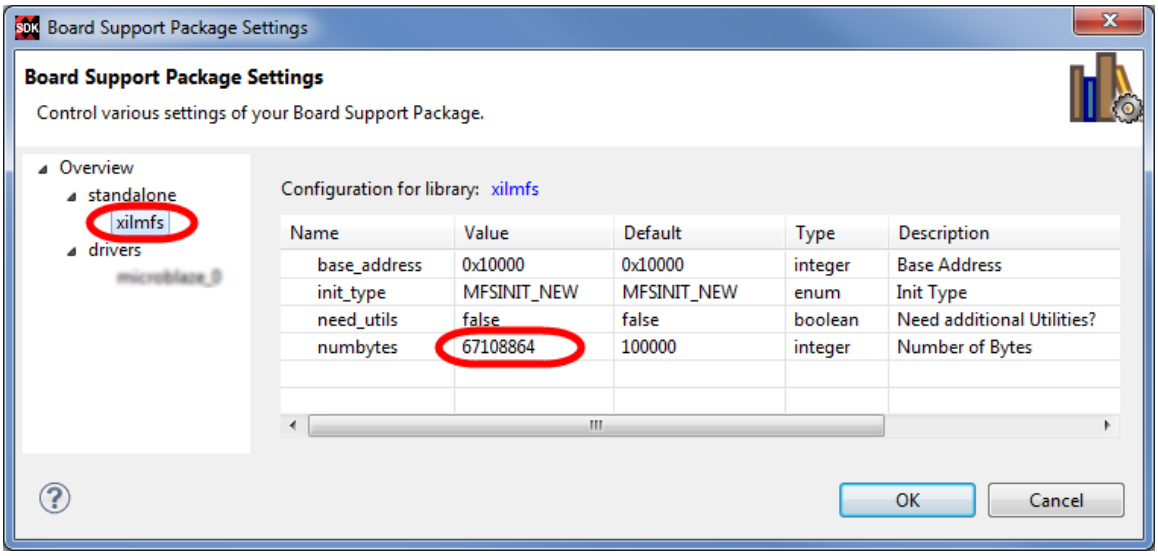


Figure 6-6: Setting File System Parameters

1-5-7. Click **OK**.

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

1-6. Add *file_system.c* to the application.

The preferred method for importing sources is shown here.

1-6-1. Expand the project named **fileSys_app** > **src** using the Project Explorer.

1-6-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

1-6-3. Select **Import** to open the Import Wizard (2).

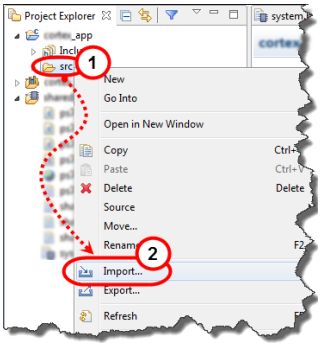


Figure 6-7: Importing a Resource File

The Import Wizard dialog box opens.

1-6-4. Expand **General** (1).

1-6-5. Select **File System** as you will be selecting individual files directly from the file system (2).

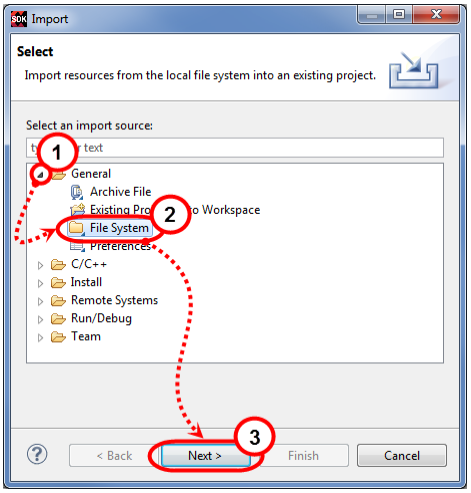


Figure 6-8: Selecting File System

1-6-6. Click **Next** to advance to specifying the files to import (3).

1-6-7. Browse to *C:\training\MemoryFileSystem\support* in the *From directory* field.

1-6-8. Select the file(s) by checking the box beside **file_system.c**.

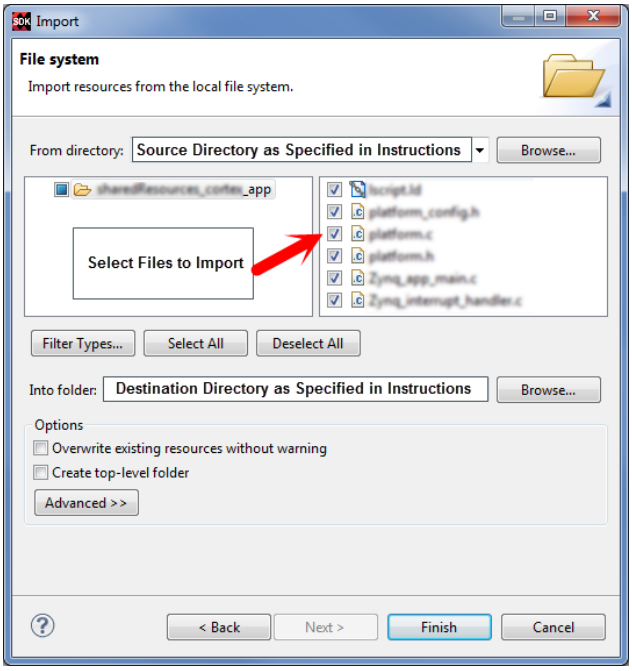


Figure 6-9: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

1-6-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

1-7. Check the linker script settings for the *fileSystem* C project.

1-7-1. Right-click the **fileSys_app** project.

1-7-2. Select **Generate Linker Script** in the Project Explorer tab.

1-7-3. Select the memory sections given below:

Zynq AP SoC users: Select **ps7_ddr_0_S_AXI_BASEADDR** as the memory area used from the Code Sections, Data Sections, and Heap and Stack drop-down lists.

MicroBlaze processor users: Select **mig_7series_0** as the memory area used from the Code Sections, Data Sections, and Heap and Stack drop-down lists.

1-7-4. Click **Generate**.

1-7-5. Click **Yes** to overwrite the existing linker script file if prompted.

1-8. Build all of the projects.

1-8-1. Force the building of all applications by selecting **Project** > **Build All**.

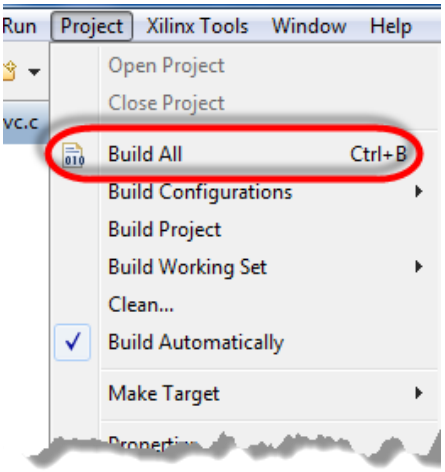


Figure 6-10: Selecting Build All

Now that the software is in place and compiled you will reference the LibXil Memory File System (MFS) documentation using the DocNav tool to better understand how the MFS functions apply.

1-9. Open the OS and Libraries document using DocNav.

1-9-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **DocNav** > **DocNav**.

Alternately, you may double-click the DocNav icon on the desktop.

The Xilinx Documentation Navigator opens.

1-9-2. Select only the **Software Development** entry under *Design Tools*.

1-9-3. Select only the **Embedded Design** entry under *Functional Categories*.

1-9-4. Select only the **Design Hubs** entry under *Document Types*.

1-9-5. Click the **Software Development KIT (SDK) for Embedded Design Hub** that was found to open the design hub.

1-9-6. Click the OS and Libraries Document Collection found under **Additional Learning Materials** > **Reference Guides**.

Note: An internet connection is required to use the DocNav.

1-9-7. Review the **LibXil Memory File System** document which is hyperlinked to the *OS and Libraries Document Collection*. Its ID is UG649.

Do not close this document, as you will be using it shortly.

Returning to the SDK tool.

1-10. Open *file_system.c* in the editor.

1-10-1. Locate **file_system.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

1-10-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

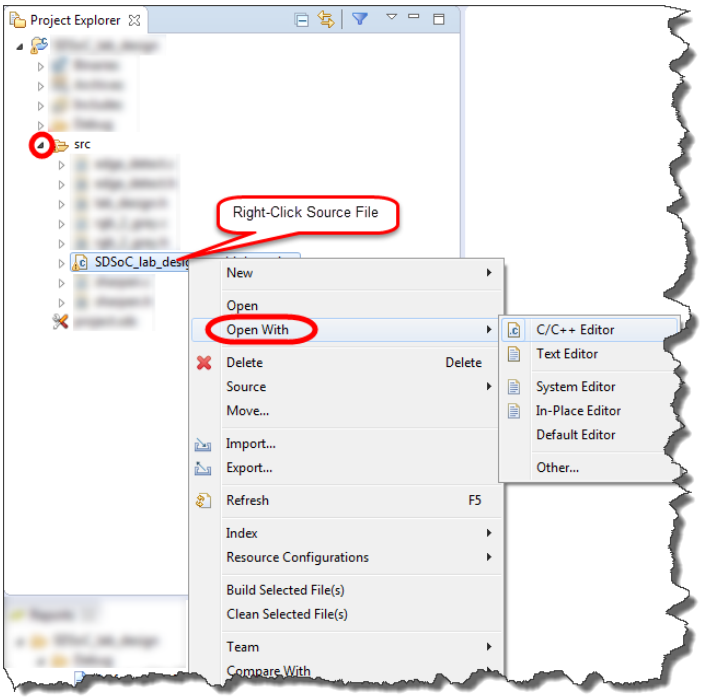


Figure 6-11: Opening a Source File via Right-Click

1-11. Review the *file_system.c* file.

1-11-1. Locate the comment "--- Initialize file system...".

File initialization is performed immediately following this line. Read the MFS function Descriptions of the function *void mfs_init_fs()* from the *oslib_rm.pdf* document.

1-11-2. Ensure that the starting(base) address of the file system memory in the *mfs_init_fs()* function call is as given below:

Zynq AP SoC users: mfs_init_fs (kNumbytes, ((char*)**0x0020FFFF**), MFSINIT_NEW);

MicroBlaze processor users: mfs_init_fs (kNumbytes, ((char*)**0x8a000000**), MFSINIT_NEW);

Similarly, find the following functions and go through the *oslib_rm.pdf* document to understand each function.

- *mfs_exists_file()*
- *mfs_delete_file()*
- *mfs_file_open()*
- *mfs_file_read()*
- *mfs_file_close()*
- *mfs_file_write()*

Refer to the *oslib_rm.pdf* library for more information on each function.

Each serial port command is assigned to specific MFS function calls as described in the procedure. Pressing a specific button will open, close, read, etc. from the file system. The results of each button press is visible on the serial port.

Question 1

How is the file system initialized? What do the arguments represent?

1-12. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

1-12-1. Select **Xilinx Tools** > **Program FPGA** or click the  icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

1-12-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

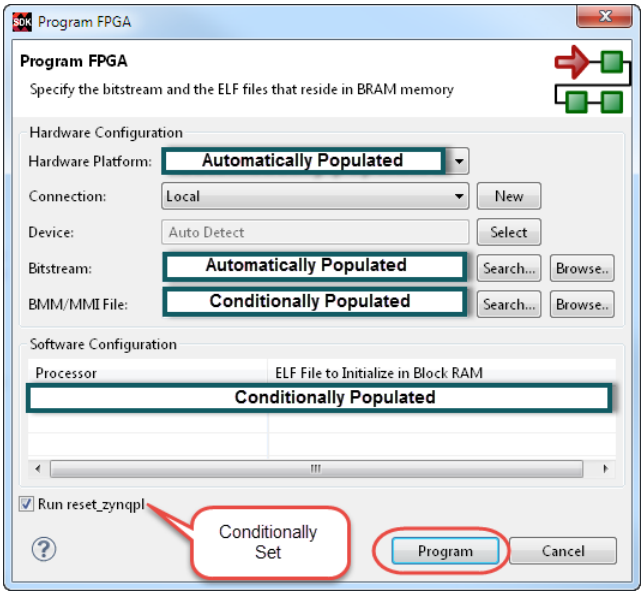


Figure 6-12: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

Running the Design Step 2

With the design now running on the board, you will test the behavior of the code.

If you are using a Zed board, please use TeraTerm or other serial port terminal as there is known incompatibility with the SDK terminal. Please configure the terminal to 115200 baud, 8 data bits, one stop bit, and no parity.

Note: If you are using the ZC702 board, follow the next two instructions. If you are using the ZedBoard, skip to instruction 2.3 below.

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

2-1. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

2-1-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

2-2. Configure the SDK Terminal.

2-2-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

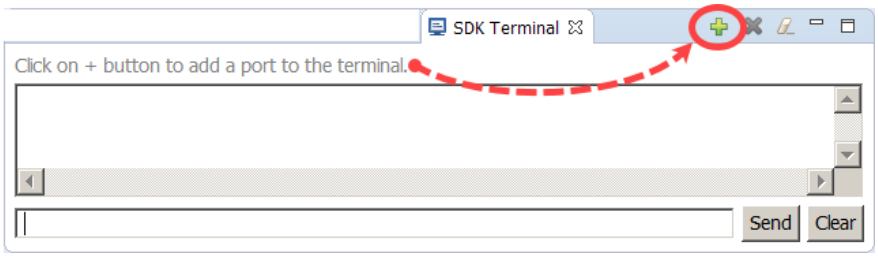


Figure 6-13: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

2-2-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

2-2-3. Set the baud rate to **115200** (2).

2-2-4. Leave the other settings at their defaults.

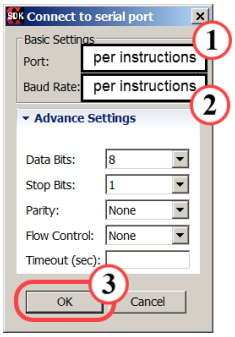


Figure 6-14: Configuring the SDK Terminal

2-2-5. Click **OK** to save these settings and begin the terminal session (3).

Note: If you are using the ZC702 board, skip this instruction.

Tera Term is a popular public domain terminal emulation program. It is capable of operating as a serial port terminal or as a telnet client. It is an alternative to using the Terminal view that is built into SDK.

2-3. Launch the Tera Term terminal program.

2-3-1. Double-click the **Tera Term** icon on the Windows desktop to launch Tera Term.

Alternatively, you can select **Start** > **All Programs** > **Tera Term** > **Tera Term**.

2-3-2. Select **Serial** as the connection (1).

2-3-3. Click the **Port** drop-down list to view the available COM ports (2).

Note: If your port is not listed, exit Tera Term, power cycle your board and re-start this step.

2-3-4. Select the COM **#** discovered in the last step (3).

Note that the ZC702 will show the Silicon Labs driver as shown in the figure below and the ZedBoard will show the Cypress driver or possibly just a USB Serial Port entry.

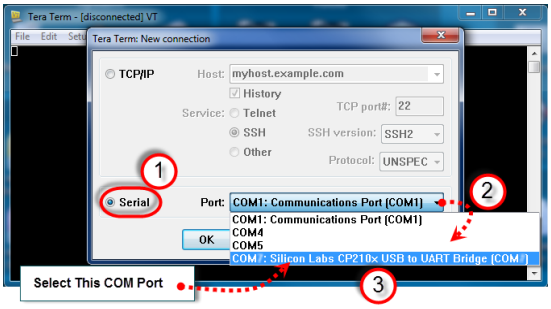


Figure 6-15: Selecting the COM Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

2-3-5. Click **OK**.

The terminal console window opens.

2-3-6. Select **Setup** > **Serial Port**.

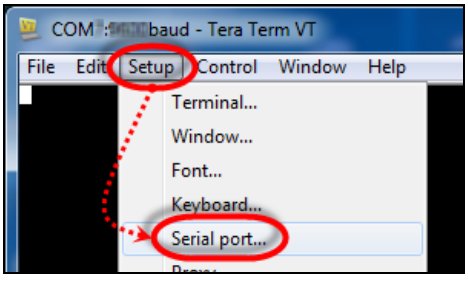


Figure 6-16: Opening the Tera Term Serial Port Setup Window

The Tera Term Serial Port Setup dialog box opens.

2-3-7. Confirm that the proper serial port has been selected (1).

2-3-8. Set the baud rate to **115200** (2).

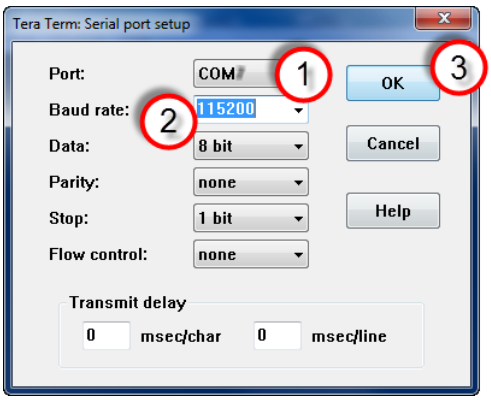


Figure 6-17: Setting the Parameters for the Serial Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

2-3-9. Click **OK** (3).

Tera Term is now configured to receive and transmit serial information to/from the evaluation board.

Configurations are a very powerful ally in setting up a run for a very specific set of criteria. Often, however, the default configuration settings are adequate and there is no need to click through a number of dialogs just to run the program.

2-4. Run *fileSys* with the default configuration settings.

2-4-1. Right-click **fileSys** in the Project Explorer to open the context menu.

2-4-2. Select **Run As** > **Launch on Hardware (System Debugger)** to immediately launch the application on the hardware.

Note: Selecting **Run As** > **Launch on Hardware (GDB)** will also work; however, this is a deprecated flow.

If an application is already running, a warning message will appear asking you to terminate the previous session. If you receive this message, click **Yes** to terminate the running application and run the new scenario.

The application runs.

--- Alternate Method ---

Select **fileSys** in the Project Explorer tab.

Once the application project is selected, you can launch the run by:

- Use the menu bar to select **Run** > **Run**.
  - Then select one of the options from the **Run As** window (this window will pop up only if there is no Run configuration).
- Press <**Ctrl** + **F11**>.

2-4-3. Click the **Run** icon ( ).

Question 2

Enter the commands into the terminal in the order that they are offered in the table below. Observe the response and record in the following table.

Button	Displayed Message	Command Meaning
C		
c		
r		
D		
d		
r		
C		
c		
r		
d		
d		

Table 1. Desired Actions and Actual Results

Question 3

What common file system functions are used?

2-5. Terminate the program.

2-5-1. Select **Window** > **Perspective** > **Open Perspective** > **Debug**.

2-5-2. Expand **System Debugger on Local fileSys_app.elf (Local)**.

2-5-3. Select the running **<CPU> (Running)**.

2-5-4. Click the **Suspend** icon ( ) to stop the running program.

2-5-5. Click the **Disconnect** icon ( ).

2-6. Modify the behavior of the code so that when the 'f' (for free) command is entered, the file system is queried and returns the number of free and used blocks.

2-6-1. Click the **C/C++** perspective view icon (C/C++) on the top of the right side of the tool.

2-6-2. Locate the comment "Block Usage Code Begin" in the *file_system.c* file.

If you do not remember how to perform this task, please refer to the **Lab Reference Guide** > **SDK Operations** > **Finding Text in a Source File**.

2-6-3. Highlight the code that begins with this line down to and including the comment "Block Usage Code End."

2-6-4. Right-click the highlighted block of code to open the pop-up menu.

2-6-5. Select **Source** > **Toggle Comment** to comment out this block of code.

2-6-6. Locate and highlight the block of code that begins with the comment "Block Usage Code Begin" and ends with the comment "Block Usage Code End."

2-6-7. Right-click the highlighted block of code to open the pop-up menu.

2-6-8. Select **Source** > **Toggle Comment** to uncomment this section of the code.

Alternately, you may use the keyboard command ctrl-/.

This block of code is the mostly complete code to implement the display of file block usage.

2-6-9. Save the **file_system.c** file.

2-7. Build all of the projects.

2-7-1. Force the building of all applications by selecting **Project** > **Build All**.

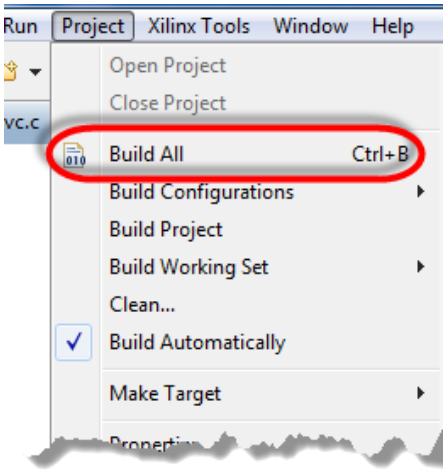


Figure 6-18: Selecting Build All

2-8. Use the MFS documentation to complete the line of code.

2-8-1. Locate the block of code that begins with the comment "// student to find and finish the proper MFS cmd to determine block usage."

2-8-2. Use the MFS documentation and update the function call to mfs_xxx_xxxxx in the following line.

2-8-3. [Optional] Update the message to the user that presents the options for this application.

2-8-4. Save the **file_system.c** file again when the unfinished code is completed.

Continue when there are no compile errors.

Please make every effort to code this yourself. If you get stuck, the necessary line of code can be found in the answers section.

2-8-5. Manually build the application by right-clicking the application to bring up the pop-up menu.

2-8-6. Select **Build Project** to rebuild all the necessary files.

2-9. Run the program.

2-9-1. Right-click the **fileSys** project in the Project Explorer tab.

2-9-2. Select **Run As** > **Launch on Hardware (System Debugger)**.

2-10. Verify the output.

2-10-1. Enter the various commands and see how the number of free and used blocks change.

Question 4

How many total MFS blocks are there? Estimate the number of bytes in a block. (Recall that the *mfs_init_fs()* function sets the number of bytes used by MFS.)

2-11. Terminate the program.

2-11-1. Follow the instructions as in instruction 2 of this step.

2-12. Exit the DocNav tool.

2-12-1. Click the 'X' in the upper right-hand corner of the tool.

2-13. Exit the SDK tool and power off the board.

2-13-1. Select **File** > **Exit** to close the SDK tool.

2-13-2. Power off the development board.

Summary

In this lab, you added the Xilinx XilMFS file system library to a software platform, ran an application that consumes and reports file system resources, and observed how many of these commands manifested. You were also able to perform various file-related operations and gain an understanding of MFS.

Answers

1. How is the file system initialized? What do the arguments represent?

   The file system is initialized on or near line 123 by the call to *mfs_init_fs*. Documentation for the MFS file system is found in the *OS and Libraries Document Collection*, which points to the LibXil MFS document.

   The first argument corresponds to the number of bytes of memory addressable to the file system. For example, this lab implements 64 MB of DDR memory. According to the code, numbytes (65,536) were allocated—this would be a relatively small file system.

   The second argument refers to the starting address in the file system memory, and the third argument describes how the file system is to be created—from scratch, from a previously loaded read/write memory, or a previously loaded read-only memory (useful for storing tables of data).

2. Press the buttons in sequence starting at the top row of the table. Observe the messages displayed on the LCD and determine the button functions.

Button	Displayed Message	Command Meaning
C	testdir1 created	Directory create
c	File created successfully	File create
D	directory deleted	Directory delete
d	testfile1 deleted successfully	File delete
r	File could not be opened for reading	Read file
C	testdir1 created	Directory create
c	File created successfully	File create
r	File read:test message	Read file
d	File deleted successfully	Delete testfile1
d	testfile1 does not exist!	Delete file, but it was already deleted

Table 2. Desired Actions and Actual Results

3. What common file system functions are used?

   mfs_init_fs( ), *mfs_exists_file( )*, *mfs_delete_file( )*, *mfs_file_open( )*, *mfs_file_read( )*, *mfs_file_close( )*, *mfs_file_write( )*, *mfs_delete_dir( )*, *mfs_create_dir( )*.

4. How many total MFS blocks are there? Estimate the number of bytes in a block. (Recall that the *mfs_init_fs()* function sets the number of bytes used by MFS.)

   There are 123 blocks. The *mfs_init_fs( )* function sets the number of bytes to 0x10000 (64K) so 65,536/123 = 532.8. Assuming the number of bytes in a block equals a power of 2, there are likely 512 bytes in a block. With 123 blocks at 512 bytes each, that is 62,976 bytes. The remaining bytes are probably used for file system overhead.

   Code snippet for the Block Usage call:

```
if ((status = mfs_get_usage(&blocks_used,&blocks_free)) == 0) {
```

Lab 7: Linker Script

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

Linker scripts guide the linker in determining how to map the various sections of code and text into physical memory. This lab demonstrates how to configure the linker script and observe increased performance when utilizing different memory sections.

Objectives

After completing this lab, you will be able to:

- Outline the default and common memory segments generated by the gcc compiler
- Edit the linker script to place code and data into different sections of memory

Introduction

Many embedded systems contain multiple types of memory—DDR, OCM, cache, block RAM, Flash, etc. How do the tools know how to use each of these memories? When a C compiler runs, it groups initialized data, uninitialized data, code, stack, heap, and other blocks together. How does it know where to put each block?

The linker script is read by the linker application and contains both a listing of the types of memory available in hardware and their locations (addresses) in the system memory map and a mapping between the sections of data and instructions to physical memory.

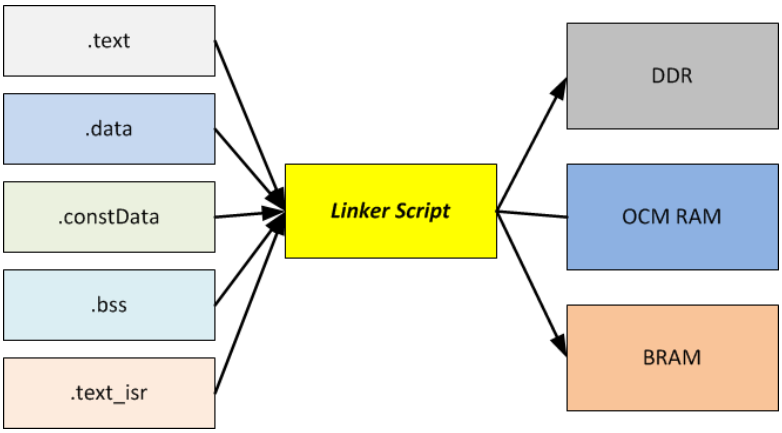


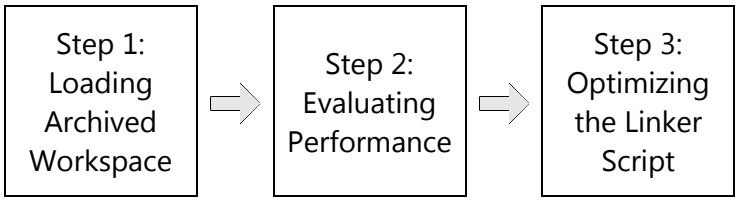
Figure 7-1: Linker Script Maps Code and Data Segments to Physical Memory

Some types of memory are better for some purposes than others. You will explore in this lab how various code and data blocks are mapped to physical memory and witness how the linker script can impact performance.

The following is a partial list of sections (and what they do) that are often generated by the gcc compiler:

.text	Contains machine instructions for user and system code. Can be subdivided into other blocks for finer grain memory positioning.
.data	Contains static data defined in the code. Static does not mean non-changing; rather it means "initialized".
.bss	Comprised of uninitialized global or static variables.
.comment	Contains version control information.
.debug_<various>	Debug information (as a result of the -g option).
.eh_frame	Contains information for frame unwinding when exceptions are handled.
.heap	Reserved for the heap; used for dynamic memory allocation.
.init	Holds instructions for initialization code (run before user's C/C++ code).
.rodata	Reserved for read-only data.
.stack	Reserved for the C/C++ function call stack.
.mmu_tbl	Contains memory look-up tables for the memory management unit (MMU).

General Flow



Opening SDK and Loading the Archived Workspace Step 1

A project has been provided for you as a starting point for your exploration of memory usage. Begin by opening the provided project.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\LinkerScript\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\LinkerScript\lab** directory and click **OK**.

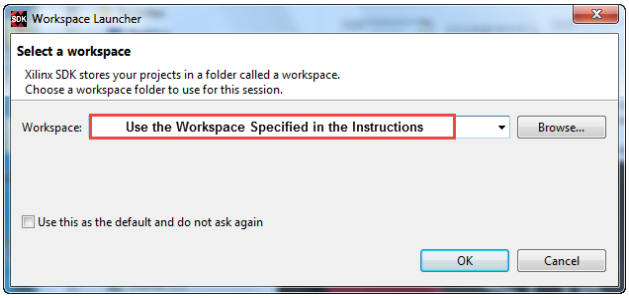


Figure 7-2: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

One or more projects can be exported as a single zipped file. This is convenient when passing projects or parts of projects to teammates or clients. Once the recipient has received this archive, the zip file must be processed and one or more projects can then be imported.

1-2. Import an existing project.

1-2-1. Select **File** > **Import** to open the Import Wizard.

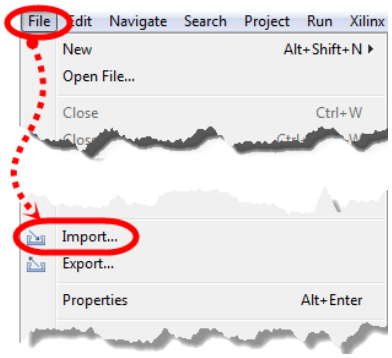


Figure 7-3: Accessing the Import Wizard

The Import dialog box opens.

1-2-2. Expand the **General** node to access the commonly used methods (1).

1-2-3. Select **Existing Projects into Workspace** as the goal is to import an existing project (2).

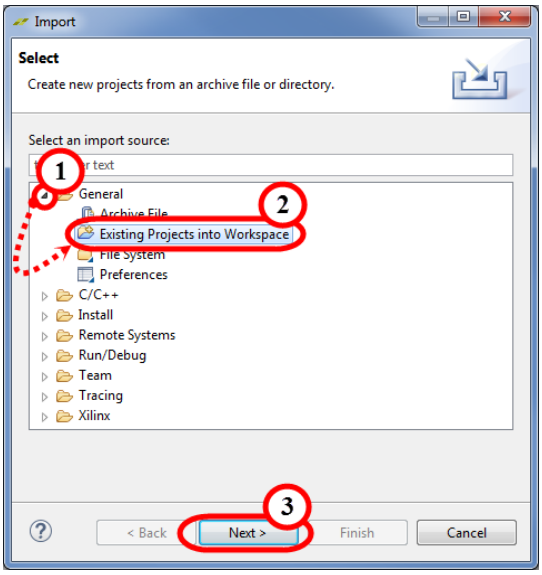


Figure 7-4: Choosing to Import an Existing Project into the Workspace

1-2-4. Click **Next** to enter project-specific data (3).

1-2-5. Select the **Select archive file** option (1).

Note that projects can be archived either as single zip files, or you can import from a directory. Typically, projects are preserved as archives as they are easier to move.

1-2-6. Click **Browse** to navigate to *C:\training\LinkerScript\support* (2).

1-2-7. Select **linkerScript_student_[zc702 | zed].zip**, which contains the archived projects.

1-2-8. Click **Open** to open the archive and list the various projects in that archive.

1-2-9. Select **all of the projects** to import (3).

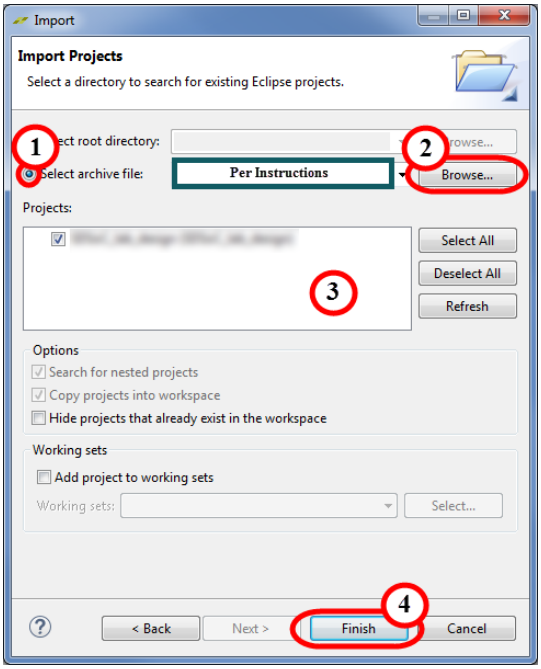


Figure 7-5: Import Settings for Archived Projects

1-2-10. Click **Finish** to perform the importing of the project (4).

There are two aspects that need to be considered when dealing with the linker script. The first is to understand the sections that were created by the compiler. The GNU tool suite offers a tool called objdump that with the appropriate arguments will display various types of information contained in the ELF (executable and load format) file produced by the tool chain (compiler, linker, loader).

A small jar file ("executable" Java code) has been provided for you as a shortcut. This file runs objdump on the ELF file that you provide and displays an alphabetically sorted list of memory sections. This is done so that you do not need to manually run the objdump tool then scan through the lengthy list of sections that is dumped when the symbol table option is provided.

1-3. Run the objDump_helper.

1-3-1. Browse to *C:\training\LinkerScript\support* using Windows Explorer.

1-3-2. Double-click **objdump_helper.jar** to launch the jar file.

Note: If the tool does not launch, you will need to confirm that there is a Java virtual machine installed on the PC.

1-3-3. Browse to *C:\Xilinx\Vivado\2016.3\tps\mingw\[highest version number]\win64.o\nt64\bin* using Windows Explorer.

1-3-4. Drag-and-drop **objdump.exe** into the *objdump@* field

1-3-5. Browse to *C:\training\LinkerScript\labslinkerScript_app\Debug* using Windows Explorer.

1-3-6. Drag-and-drop **linkerScript_app.elf** into the ELF file field of the objdump_helper program.

1-3-7. Click **Run objDump and Show Results** to read the ELF file and display the memory sections in the results text area.

Question 1

List the sections allocated for code and for data.

Now that you know what the key sections are, it is time to look at the linker script and see where these sections are mapped to in hardware.

1-4. Open the linker script associated with the *linkerScript_app* project.

1-4-1. Expand **linkerScript_app** > **src** to see all the sources associated with this project.

The linker script (named *lscript.ld*) is considered a source and therefore is located under the src directory.

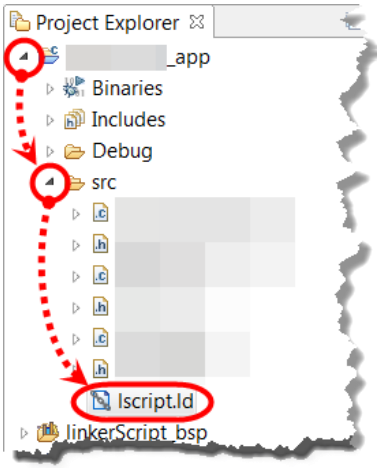


Figure 7-6: Accessing an Application's Linker Script

1-4-2. Double-click **lscript.ld** to open it in the main workspace pane.

A formatted summary of the linker script is displayed with the Summary tab (on the bottom) showing by default configuration.

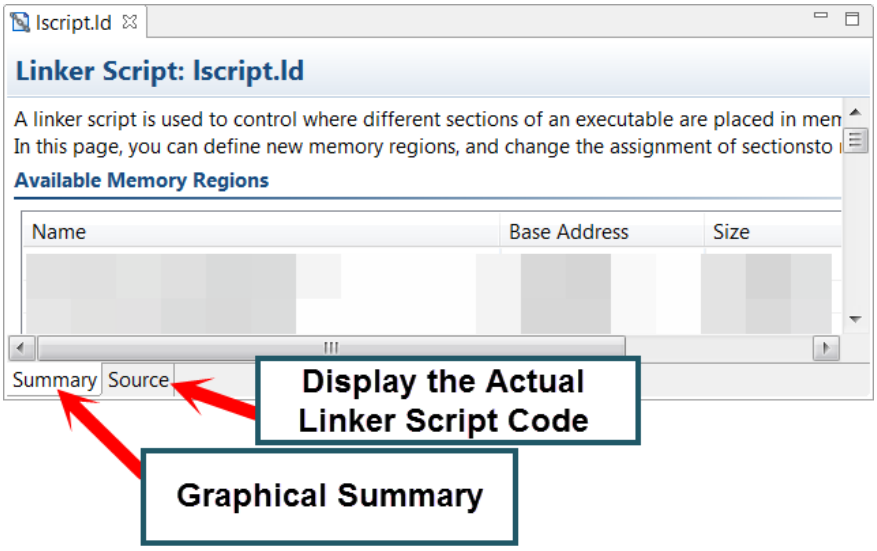


Figure 7-7: Linker Script View Showing the Default Summary

1-4-3. [Optional] Select the **Source** tab below the linker script display to review the actual linker script code.

Question 2

What are the three main sections in the linker script display?

Question 3

Where are all the memory regions mapped to?

Evaluating Performance Based on the Provided Linker Script Step 2

The provided design uses a timer to measure performance. It first measures the amount of time it takes to perform a single pass of a *for* loop (which is used later to obtain a more accurate measurement of the time it takes to call a function). After this measurement is made, it calls a function in another loop and subtracts out the loop time. This provides a pretty good measurement for how long it takes to run the function.

The premise is that by moving this function and its associated data from DDR memory to faster/lower-latency memories that this function will take less time to execute. You will now test this premise by measuring the current configuration as a baseline, then moving the various data and instruction sections to different memories to see if there is any performance improvement.

Begin by understanding how this code works.

2-1. Open *linkerScript_main.c* in the editor.

2-1-1. Locate **linkerScript_main.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

2-1-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

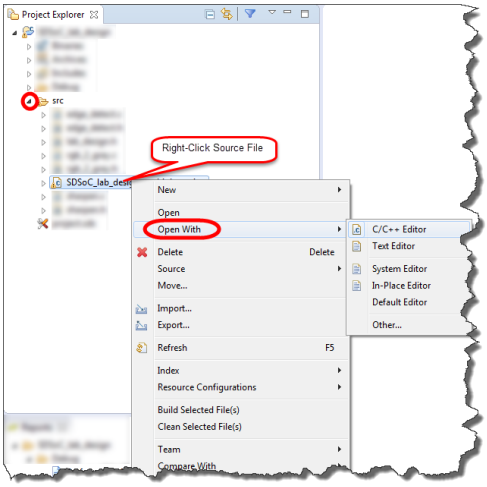


Figure 7-8: Opening a Source File via Right-Click

2-1-3. Locate the comment "*measure how long it takes to do a loop*" to locate the code that measures loop duration.

2-2. Find *measure how long*.

The find/replace operation is accessible through both a click sequence and a keyboard shortcut.

2-2-1. Select **Edit** > **Find/Replace** or press <**Ctrl** + **F**>.

The Find/Replace dialog box opens.

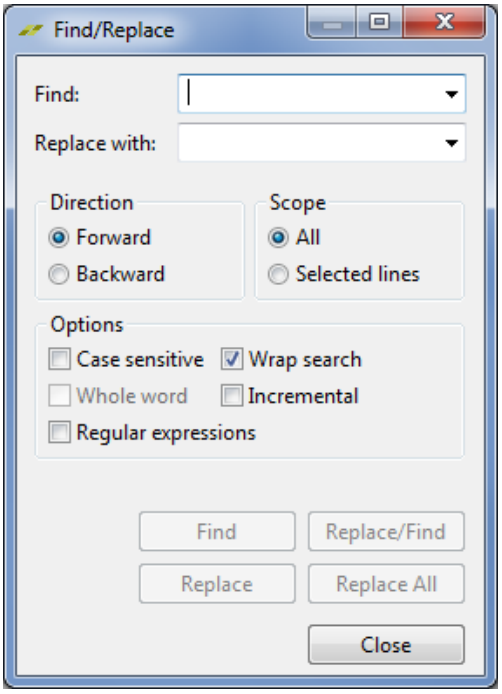


Figure 7-9: Default Find/Replace Dialog Box

2-2-2. Enter **measure how long** in the Find field.

2-2-3. Click **Find** or press <**Enter**> to find the next occurrence of *measure how long*.

2-2-4. Continue clicking **Find** or pressing <**Enter**> until you locate the specific instance that you are looking for.

With the **Wrap search** option enabled, the file is treated as a continuous loop and the find operation will jump to the next occurrence at the top of the file (when searching forwards) or at the bottom of the file (when searching backwards). A "ding" sound is made when the search wraps around.

If you are looking for text within a specific region of the code you would first highlight the region to perform the search in, then launch the Find/Replace function as described in this topic.

2-2-5. Click **Close** to close the Find/Replace dialog box.

The next few lines measure and display the loop overhead.

Question 4

How is the loop overhead measured?

2-2-6. Look at the next section of code below the comment "*// measure how long it takes to call a function*".

Question 5

How is the time that is needed to execute a function computed?

Question 6

Is this an accurate way to measure time?

Prior to launching the application, the board must be configured with the bitstream. Make sure that the hardware is properly cabled. Connect the cables (USB-UART and USB programming cables) and power on the board.

2-3. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

2-3-1. Select **Xilinx Tools** > **Program FPGA** or click the icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

2-3-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

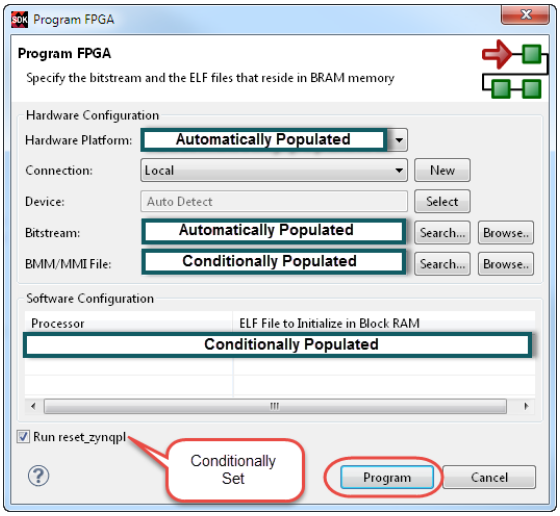


Figure 7-10: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

Since the application will report these times through the serial port, the Terminal tab must be configured to the proper settings and enabled (connected).

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

2-4. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

2-4-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

2-5. Configure the SDK Terminal.

2-5-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

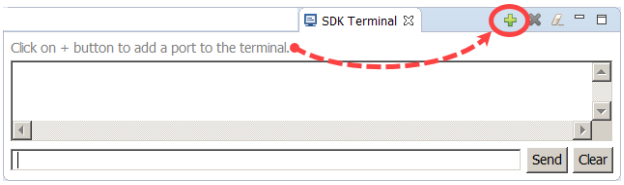


Figure 7-11: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

2-5-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

2-5-3. Set the baud rate to **115200** (2).

2-5-4. Leave the other settings at their defaults.

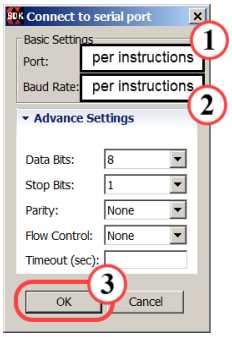


Figure 7-12: Configuring the SDK Terminal

2-5-5. Click **OK** to save these settings and begin the terminal session (3).

The linker script controls where various pieces of the software reside in physical memory and how the physical memory is partitioned for use by the application.

2-6. Create a custom linker script for a C/C++ application.

2-6-1. Right-click **linkerScript_app**.

2-6-2. Select **Generate Linker Script**.

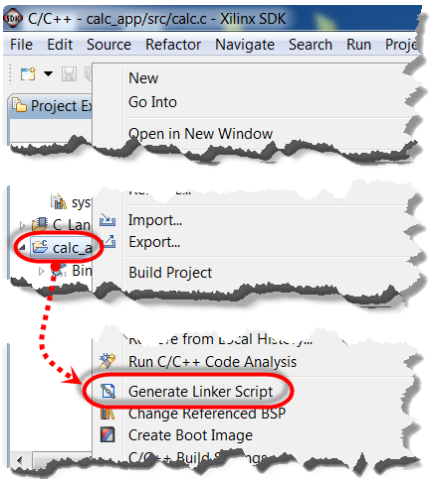


Figure 7-13: Accessing the Generate Linker Script Capability

The Generate Linker Script dialog box appears.

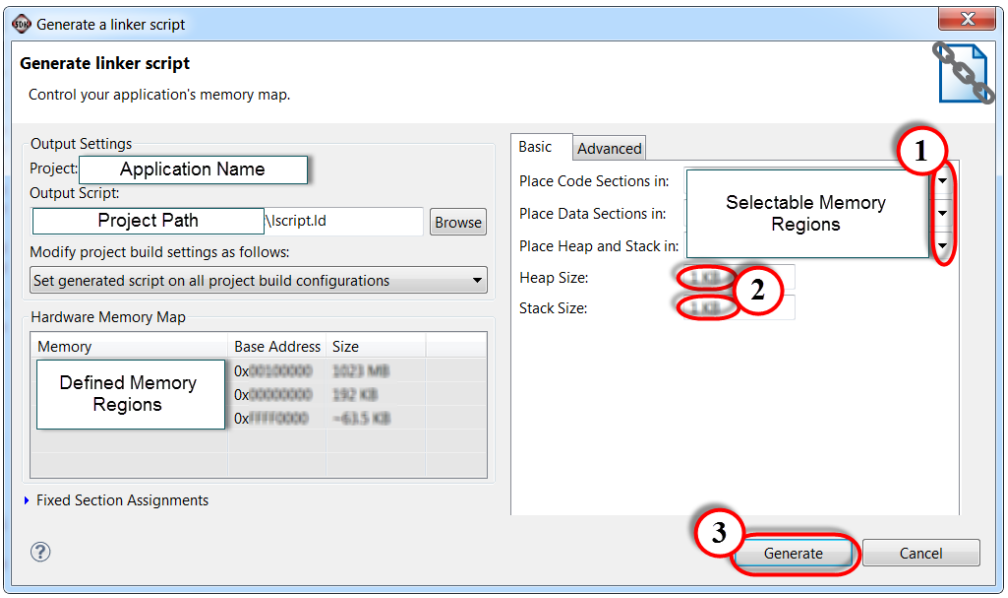


Figure 7-14: Common Sections of the Generate Linker Script Dialog Box

From here, you can select where your code, data, and help sections reside in physical memory, as well as assigning how much space is allocated to your heap and stack.

2-6-3. **Zynq AP SoC users:** Verify that **ps7_ddr_0_S_AXI_BASEADDR** is selected as the memory area used from the Code Sections, Data Sections, and Heap and Stack drop-down lists (1).

2-6-4. Leave the Stack and Heap sizes at 1 KB (2) as the user code does not require any heap.

Note that some library functions may require the heap but since there are very few function calls for the application in this lab, a 1K stack size is sufficient.

2-6-5. Click **Generate** to build the linker script.

2-6-6. Click **Yes** if you are asked to overwrite the existing linker script.

The new linker script will be generated.

Configurations are a very powerful ally in setting up a run for a very specific set of criteria. Often, however, the default configuration settings are adequate and there is no need to click through a number of dialogs just to run the program.

2-7. Run *linkerScript_app* with the default configuration settings.

2-7-1. Right-click **linkerScript_app** in the Project Explorer to open the context menu.

2-7-2. Select **Run As** > **Launch on Hardware (System Debugger)** to immediately launch the application on the hardware.

Note: Selecting **Run As** > **Launch on Hardware (GDB)** will also work; however, this is a deprecated flow.

If an application is already running, a warning message will appear asking you to terminate the previous session. If you receive this message, click **Yes** to terminate the running application and run the new scenario.

The application runs.

--- Alternate Method ---

Select **linkerScript_app** in the Project Explorer tab.

Once the application project is selected, you can launch the run by:

- Use the menu bar to select **Run** > **Run**.
  - Then select one of the options from the **Run As** window (this window will pop up only if there is no Run configuration).
- Press <**Ctrl** + **F11**>.
- Click the **Run** icon ( ).

Question 7

Record the output of the application (function call overhead) in the **DDR (Zynq AP SoC) Only Configuration Time (In Counts)** column in the table below.

Function Call Number	DDR Only (Zynq AP Soc) Configuration Time (In Counts)	OCM RAM (Zynq AP SoC BRAM) Time (In Counts)	Zynq AP Soc Hybrid Memory Usage (In Counts)
1			
2			
3			
4			

Optimizing the Linker Script and Re-Evaluating Step 3

For the purpose of this lab, the absolute values of these calls are not of concern (you are not trying to achieve a specific performance); rather, the relative performance between running the entire application from DDR and other configurations is.

You will begin by changing the linker script to place the data and code sections in a memory more local to the processor than the DDR.

3-1. Change the mappings of all of the sections of code from DDR to lower latency memory.

This will cause the loader to place all the sections (for both instructions and data) into the OCM RAM in the PS.

3-1-1. Right-click **linkerScript_app**.

3-1-2. Select **Generate Linker Script** from the context menu.

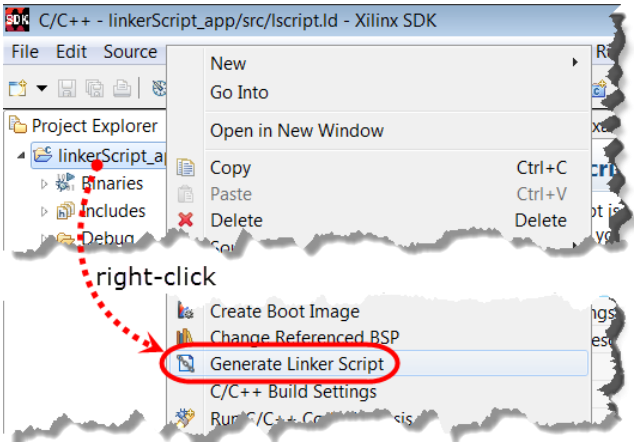


Figure 7-15: Starting the Generate Linker Script Wizard

The Linker Script Wizard opens.

The Advanced tab provides fine granularity section placement; however, this granularity is not required for the next instruction so, for convenience, you will perform the following tasks from the Basic tab.

3-1-3. **Zynq AP SoC users:** Select **ps7_ram_0_S_AXI_BASEADDR** from the Code, Data, and Heap and Stack sections drop-down lists.

3-1-4. Leave the heap and stack size at their default 1KB.

3-1-5. Click **Generate** to build the new linker script.

3-1-6. Click **Yes** to overwrite the existing linker script.

Observe the effects of this change.

Configurations are a very powerful ally in setting up a run for a very specific set of criteria. Often, however, the default configuration settings are adequate and there is no need to click through a number of dialogs just to run the program.

3-2. Run *linkerScript_app* with the default configuration settings.

3-2-1. Right-click **linkerScript_app** in the Project Explorer to open the context menu.

3-2-2. Select **Run As** > **Launch on Hardware (System Debugger)** to immediately launch the application on the hardware.

Note: Selecting **Run As** > **Launch on Hardware (GDB)** will also work; however, this is a deprecated flow.

If an application is already running, a warning message will appear asking you to terminate the previous session. If you receive this message, click **Yes** to terminate the running application and run the new scenario.

The application runs.

--- Alternate Method ---

Select **linkerScript_app** in the Project Explorer tab.

Once the application project is selected, you can launch the run by:

- Use the menu bar to select **Run** > **Run**.
  - Then select one of the options from the **Run As** window (this window will pop up only if there is no Run configuration).
- Press <**Ctrl** + **F11**>.
- Click the **Run** icon ( ).

Question 8

Fill in the **OCM RAM (Zynq AP SoC BRAM) Time (In Counts)** column in the previous table (function call overhead).

Question 9

What was the net effect of moving code and data out of DDR and placing it in lower latency memory?

Question 10

Will this approach always work?

Question 11

How can you force the compilation tools to place functions that must have low latency into memories that support lower latency?

Here you will mark the C code to indicate which pieces of data and code can be moved to different sections in memory.

If you closed the editing tab, you will need to reopen the source file as described above.

Attributes are used to direct the compiler and linker/loader to treat certain functions and variables differently. The generic format for an attribute is "__attribute__((<attribute to apply>))";

Attributes for the GNU C Compiler are covered in detail on the GNU website. For your optional exploration, here are links to the site where attributes are discussed:

- For functions: gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html.
- For variables: gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html.
- For types: gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html.

You will use attributes to place a function and variables into different segments.

3-3. Apply a new section attribute for *targetFunction()*.

3-3-1. Find the **targetFunction()** prototype below the "*function prototypes*" comment.

3-3-2. Change the line to read:

```
int targetFunction(int x) __attribute__((section(".text_lowLatency")));
```

This will cause the compiler to create a new section named ".text_lowLatency" and the function will be placed into that section. The name of the section is not important other than it should provide you with some meaning. Since this function will be called frequently, it will be placed into low latency memory.

3-4. Apply a new section attribute for the table of constants named *constTbl[][]*.

3-4-1. Find the **constTbl** variable definition below the "*constant table*" comment.

3-4-2. Change the lines to read:

```
int constTblWidth __attribute__((section(".data_constTbl"))) = 3;
int constTblLength __attribute__((section(".data_constTbl"))) = 2;
int constTbl[2][3] __attribute__((section(".data_constTbl"))) = {{1,3,5}, {2,4,6}};
```

This will cause the compiler to create a new section named ".data_constTbl" and will place these three variables into this section. As with the function segment, the segment name is not important other than it should be meaningful to you and should not duplicate an existing segment name.

3-5. Save and compile your open source file.

3-5-1. Press <**Ctrl** + **S**>, click the **Save** icon ( ) or the **Save All** icon ( ), or select **File** > **Save**.

Each time a file is saved, it is automatically rebuilt. You can monitor the behavior in the console view. Alternately, you can force all sources to be rebuilt by selecting **Project** > **Build All**.

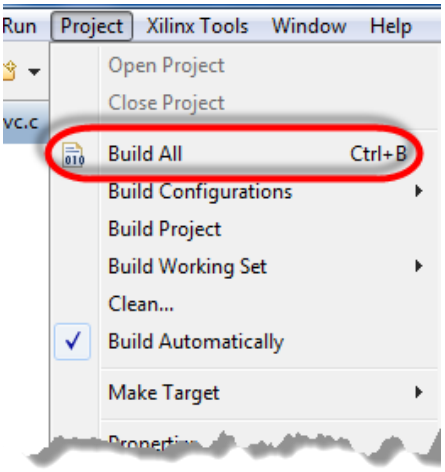


Figure 7-16: Selecting Build All

3-6. Check to see if the new sections have been properly created.

3-6-1. Return to the objdump_helper program and click **Run objDump and Show Results** to reread the ELF file and display the memory sections in the results text area.

Question 12

Do the new segments show up?

3-7. Move all the sections back to DDR memory.

This will cause the loader to place all the sections (for both instructions and data) into the DDR RAM as this is typically where larger programs must reside.

3-7-1. Right-click **linkerScript_app** to access the options for this project.

3-7-2. Select **Generate Linker Script** from the context menu to open the Linker Script Wizard.

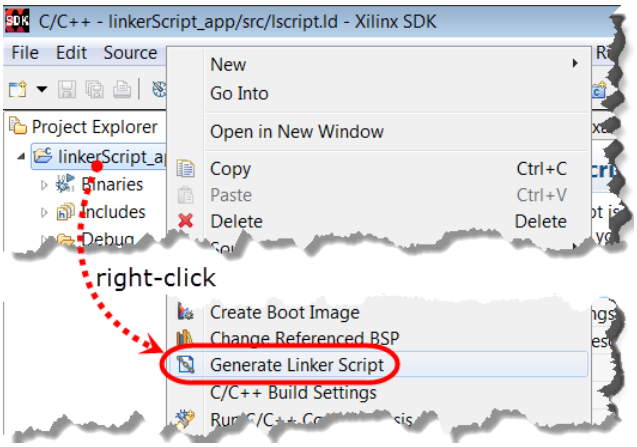


Figure 7-17: Starting the Generate Linker Script Wizard

The Linker Script Wizard opens.

Since the gross-level placement can be made from the Basic tab, you will perform the following tasks from the Basic tab.

3-7-3. **Zynq AP SoC users:** Select **ps7_ddr_0_S_AXI_BASEADDR** from the Code, Data, and Heap and Stack drop-down lists.

3-7-4. Leave the heap and stack size at their default 1KB.

3-7-5. Do NOT click **Generate**, as there are a few more changes required for this linker script.

3-8. Place *targetFunction()* and the constant table into OCM RAM for the Zynq AP SoC.

So far you have been working in the Basic tab to change memory targets. The Advanced tab offers additional options for finer granularity segment placement.

3-8-1. Select the **Advanced** tab (1).

3-8-2. Click **Add Section** under the Code Section Assignments area (2).

Here you will add the name of the new code section that you created for *targetFunction()*.

3-8-3. Enter **.text_lowLatency** in the section name field, which is the name of the segment for the function that you want low latency for.

3-8-4. Click **OK** to register the new name (3).

The new name now appears in the Code Section Assignments area.

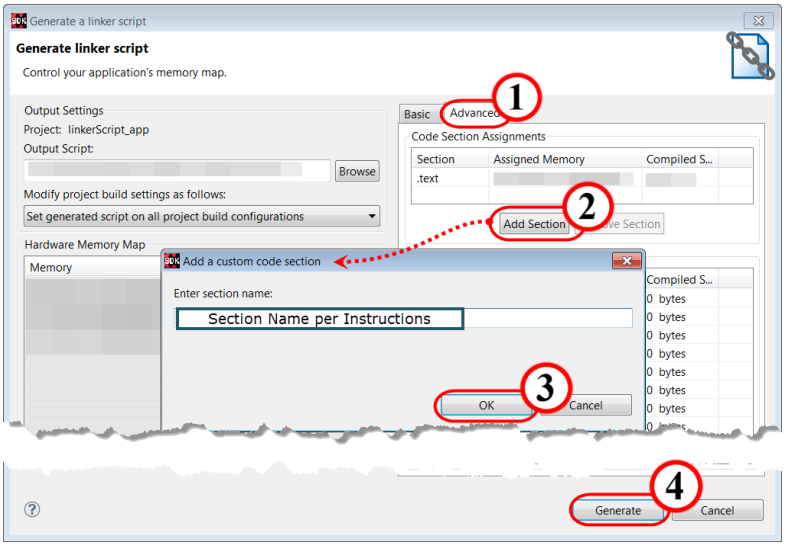


Figure 7-18: Adding a Section to the Linker Script

3-8-5. **Zynq AP Soc users:** Use the drop-down list in the .text_lowLatency row to select **ps7_ram_0_S_AXI_BASEADDR**.

3-9. Place the constant table and supporting variables into OCM RAM for the Zynq AP SoC.

While the constant table is not actually an executable (code) segment, it can be entered into the Code Section Assignments area anyway.

3-9-1. Click **Add Section** under the Code Section Assignments area.

Here you will add the name of the new data section that you created for the constant table and supporting variables.

3-9-2. Enter **.data_constTbl** in the section name field, which is the name of the segment for the function that you want low latency for.

3-9-3. Click **OK** to register the new name.

The new name now appears in the Code Section Assignments area.

3-9-4. **Zynq AP Soc users:** Use the drop-down list in the .constTbl row to select **ps7_ram_0_S_AXI_BASEADDR**.

3-10. Generate the new linker script.

3-10-1. Click **Generate** to build the new linker script.

3-10-2. Click **Yes** in the dialog box asking if the existing script file should be overwritten since a linker script already exists.

3-11. Rerun the application to observe the new performance of the hybrid memory usage version of the linker script.

3-11-1. Select **Run** > **Run** to relaunch the application.

3-11-2. Select the **Terminal** tab to see the results of the run.

Question 13

Fill in the **Zynq AP SoC Hybrid Memory Usage (In Counts)** column in the previous table (function call overhead).

Question 14

What kind of performance difference do you observe?

Summary

The configuration of the linker script can have a huge impact on the overall performance of a system.

You began by running the entire application from DDR memory. You then reconfigured the linker script to run from fast, local memory and saw significant performance improvements. But, because it is rare that a real application is small enough to fit in the local memory, you tested a compromise—leaving the bulk of the code in DDR and moving the frequently used data and function into local memory. When you reran this configuration, you saw that the performance was very close to that of the local memory use.

Note that the caches were disabled. It is left as an exercise to the student to comment out the cache disables and move all of the application back into DDR memory and rerun the tests.

Answers

1. List the sections allocated for code and for data.
   - Code sections: .text
   - Data sections: .bss, .data, .init, .init_arry, .rodata
   - Other important sections: .stack, .heap
2. What are the three main sections in the linker script display?

   The Available Memory Regions section shows the physical memory available in this hardware design.

   The Stack and Heap Sizes section lists the number of bytes associated with each of these sections.

   The Section to Memory Region Mapping section shows the mapping between the code sections and the physical memory.
3. Where are all the memory regions mapped to?

   All the sections are mapped to DDR memory.
4. How is the loop overhead measured?

   The timer is read and a loop sequence is started. At the end of the loop sequence, time is read again. This difference between the start and the end is measured and rounded off. Since calling the time reading routine itself takes time, this time is removed from the calculation.
5. How is the time that is needed to execute a function computed?

   In a similar way to how the loop overhead was measured. A time measurement is taken and the function is called four times. The difference in time is recorded for each call. The overhead due to the *for* loop is subtracted out.
6. Is this an accurate way to measure time?

   It is for the purpose of this lab. The difficulty comes in that the first time the function is called it must be pulled into the cache from DDR. The subsequent calls will run the function from cache. This is why the time for each call is independently recorded, so that you can see the impact of the cache on performance.

7. Record the output of the application (function call overhead) in the **DDR Only (Zynq AP SoC) Configuration Time (In Counts)** column in the table below. (Your numbers may not match exactly.)

Function Call Number	DDR Only (Zynq AP SoC) Configuration Time (In Counts)	OCM RAM (Zynq AP SoC BRAM) Time (In Counts)	Zynq AP Soc Hybrid Memory Usage (In Counts)
1	151		
2	122		
3	127		
4	127		

8. Fill in the **OCM RAM (Zynq AP SoC BRAM) Time (In Counts)** column in the previous table (function call overhead). (Your numbers may not match exactly.)

Function Call Number	DDR Only (Zynq AP SoC) Configuration Time (In Counts)	OCM RAM (Zynq AP SoC BRAM) Time (In Counts)	Zynq AP SoC Hybrid Memory Usage (In Counts)
1	151	55	
2	122	53	
3	127	53	
4	127	53	

9. What was the net effect of moving code and data out of DDR and placing it in this lower latency memory?

   You should have seen an improvement in the performance on the order of x2.5 for the Zynq AP SoC system.

10. Will this approach always work?

    Unfortunately not. The presence of caches definitely helps with functions and data that is frequently used, but cache behavior is non-deterministic and may suffer for long latency just at the wrong moment. If the application is small enough, it can be run from these faster memories; however, this is often not the case—local (low-latency) memory is expensive and there is rarely enough of it.

    Time to compromise—move everything back to DDR except the functions and data that need the low latency (such as frequently used data and functions and interrupt service routines).

11. How can you force the compilation tools to place functions that must have low latency into memories that support lower latency?

    Specific functions and variables can be *marked* and placed into their own segments. These segments can then be mapped to the desired memories.

12. Do the new segments show up?

    The answer should be yes. If not, return to SDK and make sure that there were no errors and, if necessary, for a rebuild of the application.

13. Fill in the **Zynq AP SoC Hybrid Memory Usage (In Counts)** column in the previous table (function call overhead). (Your numbers may not match exactly.)

Function Call Number	DDR Only (Zynq AP SoC) Configuration Time (In Counts)	OCM RAM (Zynq AP SoC BRAM) Time (In Counts)	Zynq AP SoC Hybrid Memory Usage (In Counts)
1	151	55	122
2	122	53	115
3	127	53	119
4	127	53	115

14. What kind of performance difference do you observe?

    You should have noticed much better performance than the pure DDR design and perhaps almost as good as the local memory.

Lab 8: Software Interrupts

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

Interrupts provide a low-latency response to events. This lab demonstrates how to replace a software timing loop with an interrupt-driven timer.

Objectives

After completing this lab, you will be able to:

- Navigate Xilinx processor device services documentation
- Use the BSP processor interrupt services
- Navigate device driver API documentation
- Describe interrupt controller and timer device driver services

Introduction

There are two basic ways for software and hardware to communicate information regarding events: polling and interrupting. Polling is the practice of periodically, at the software's convenience, reading from a device to see if anything has happened. While this technique is acceptable for certain types of applications, there are times that the hardware needs to get the software's attention right away. This is done by using an interrupt.

An interrupt is simply a line tied to the processor that, when asserted, causes the processor to stop the normal flow of execution and run a special function known as an interrupt handler or interrupt service routine (ISR). This interrupt handler is a small, user-written, specialized piece of code that reacts in a prescribed way to the event that caused the interrupt.

This lab will convert a stopwatch application from a software-based timing loop (polled) to an interrupt-driven timer-based one. The application program has been mostly written for you and includes usage of the general interrupt controller (GIC) and snoop control unit (SCU) timer peripherals for the Zynq® All Programmable SoC device and an interrupt controller and AXI timer for MicroBlaze™ processor-based designs.

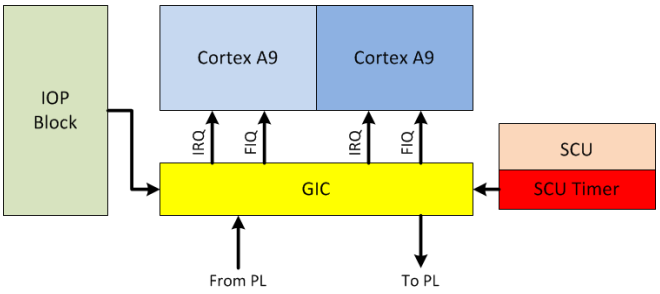


Figure 8-1: Zynq AP SoC Interrupt Block Design for this Lab

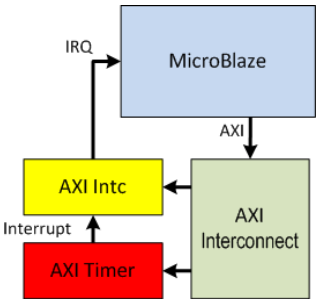


Figure 8-2: MicroBlaze Processor Interrupt Block Design for this Lab

The application program performs the following regarding the interrupt:

- Initializes the processor interrupts
- Initializes the interrupt controller
- Registers the interrupt controller interrupt service routine (ISR) with the processor interrupt data structure
- Registers the timer ISR with the interrupt controller interrupt data structure

You will be required to search BSP processor services documentation to find and use the correct processor service calls that initialize the processor interrupt data structure and enable interrupts. You will also examine the provided API documentation for the GIC and SCU timer device driver services.

General Flow

Creating an SDK Software Application Step 1

You will begin this step by launching the Xilinx Software Command Line Tool to run a short Tcl script that creates a hardware project, a BSP project, and a blank application. The script also imports the provided C source code. You will then open the SDK tool to view what was created. This step concludes when you modify the linker script and set custom compiler options.

You will use the general-purpose (shared) workspace directory to store all the applications that you develop. If there are other projects in this workspace you will see how multiple projects can be managed in a single Xilinx SDK workspace.

1-1. Launch the Xilinx Software Command Line Tool (XSCT).

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx Software Command Line Tool 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

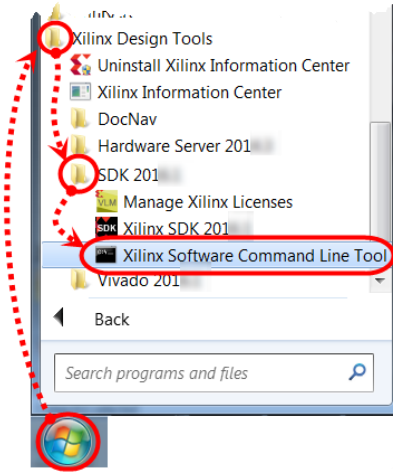


Figure 8-3: Launching XSCT

The Xilinx Software Command Line Tool opens.

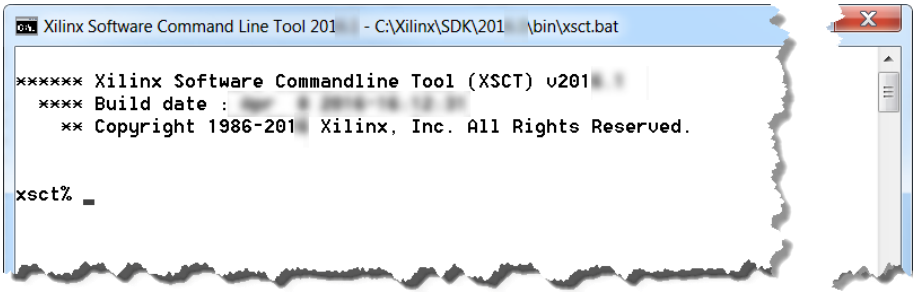


Figure 8-4: Xilinx Software Command Line Tool

You can now enter Tcl commands into the tool.

1-2. Run the provided Tcl script to build the SDK workspace

1-2-1. Enter the following command to change the directory to the location of the Tcl script you want to run:

```
cd c:/training/SWinterrupts/support
```

1-2-2. Enter the following command to run the script:

```
source SWinterrupts_builder.tcl
```

This will load the environment with the Tcl *procs*.

1-2-3. Enter the following command to specify the board type, using either ZC702 or Zed as the argument to the `use` proc:

```
use [zc702 | zed]
```

1-2-4. Enter the following command to build the projects using the Tcl proc `buildProject`:

```
buildProjects
```

You will notice errors and warnings. You can ignore these for now.

1-2-5. Click the red 'X' in the upper right-hand corner of the window to close the Tcl environment.

The Tcl script just saved you the time and effort of creating a workspace, a hardware specification project, a BSP project, an application project, and adding sources to the application project. You will continue from here by running the SDK tool and proceeding with the normal flow of application development and debugging.

1-3. Launch the SDK tool and set the workspace.

1-3-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-3-2. Enter **C:\training\SWinterrupts\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\SWinterrupts\lab** directory and click **OK**.

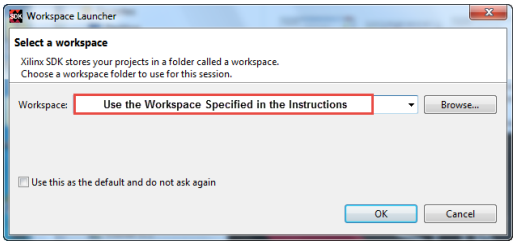


Figure 8-5: Setting Up the Workspace Environment Path

1-3-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-3-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

When running the Tcl script, you might see some error messages depending on the hardware platform that you are using. These errors are the result of undefined constants. You will address these now.

Since the source code that you just imported is used for several different processors and peripheral configurations, you will set compile SYMBOLS to direct the compiler to include the right instructions for both the processor and LCD peripheral.

Use the following table to determine which options you will select, then use the subsequent instructions to set at least one processor and, optionally, enable the LCD.

Symbol Name	Purpose
CORTEX_A9	Defined when the processor target is the Cortex™-A9 processor (Zed and ZC702 platforms)
MICROBLAZE	Defined when the processor target is the MicroBlaze processor (on the Zed and ZC702 platforms)

Compiler options describe the designer's intentions and goals to the compiler. These options directly affect how the object file is constructed. Here you will visit some of the most commonly used options.

1-4. Access the compiler options for the application.

1-4-1. Right-click the application to change or verify compiler options for and select **C/C++ Build Settings**.

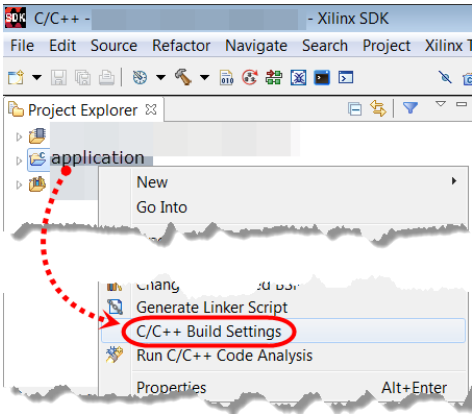


Figure 8-6: Accessing the Compiler Settings for a Specific Application

The C/C++ Build Settings dialog box opens.

1-4-2. Click **Settings** to access the specific settings for the compiler.

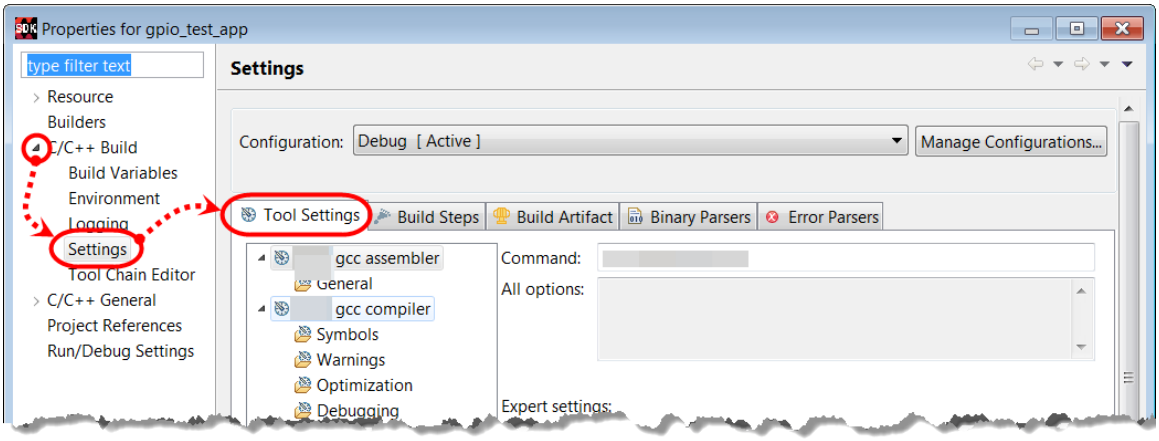


Figure 8-7: Accessing the Compiler Settings

1-5. Compiler symbols enable you to define symbols for the pre-processor. By defining symbols here, no changes need to be made to the source code. The Defined Symbols entry enables you to add, remove, edit, and reorder your symbols.

Remember that #ifdef only tests to see if a preprocessor symbol has been defined or not, and #if can test against specific values (e.g., assigning a value to a symbol can be done like this: NEW_SYMBOL 123)

1-5-1. Click **Symbols** under the compiler entry under Tool Settings.

1-5-2. Click the green **+** icon to create a new symbol.

1-5-3. Enter **CORTEX_A9 or MICROBLAZE** into the Enter Value dialog box.

1-5-4. Click **OK** to complete the creation of this new symbol.

The above steps can be repeated as necessary to create as many symbols as you need.

1-5-5. Click **Apply**.

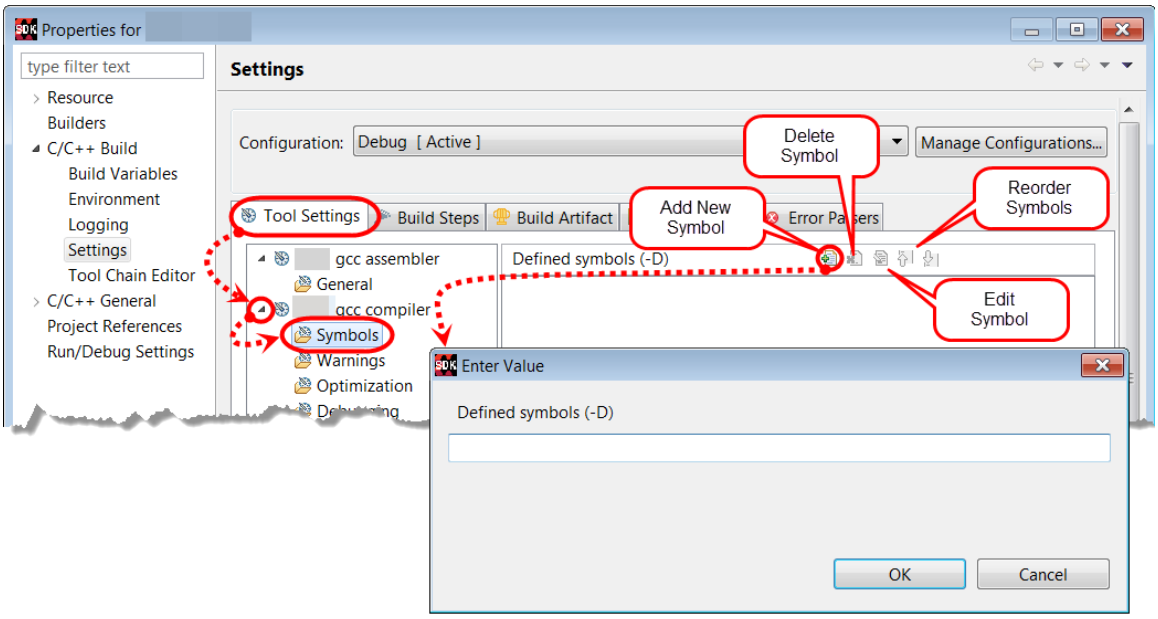


Figure 8-8: Creating a New Symbol

1-6. The Optimization tab enables you to select the level of optimization needed to meet your timing requirements for this application. Note that the optimization should be set to Zero (-O0) for debugging; otherwise, the one-to-one correspondence between source code and executing code is lost.

Remember that the levels of optimization listed in the pull-down menu are actually collections of specific optimization flags. Refer to the manual for a listing of which flags are enabled for each optimization level. If you are not happy with the pre-defined optimization levels, you can add your own optimization flags to the *Other optimization flags* text field. Remember that optimizations can change the way your code operates!

1-6-1. Select the **Optimization** tab.

1-6-2. Select your default level of optimization from the Optimization Level drop-down list.

1-6-3. Add any additional flags to the Other optimization flags field.

1-6-4. Click **Apply**.

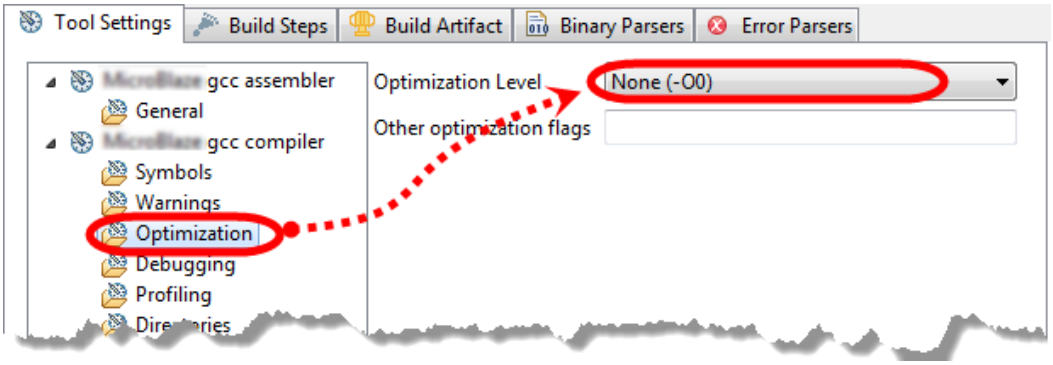


Figure 8-9: Setting Optimization Levels

1-6-5. Click **OK** to save your settings and exit.

1-7. The Debugging tab enables you to select the level of debug information to debug the application. The debug levels are - g1 (minimal debug information), -g (default debug information) and -g3 (maximum debug information).

1-7-1. Select the **Debugging** tab.

1-7-2. Select **Maximum (-g3)** from the Debug level drop-down list.

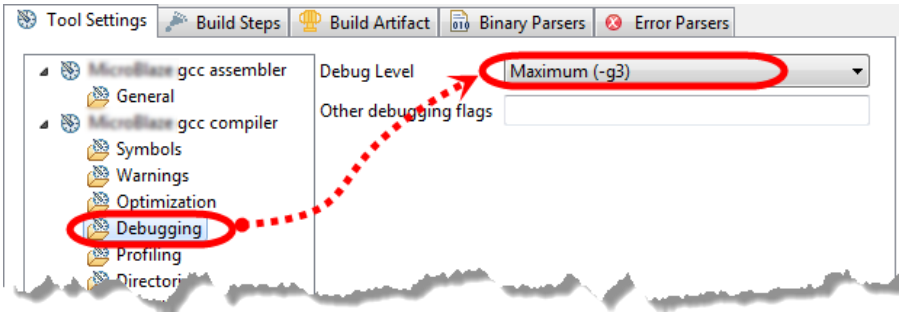


Figure 8-10: Setting Debug Level

1-7-3. Click **OK**.

The application is rebuilt. A successful build is indicated when the program size is returned.

One byproduct of creating projects using the Xilinx Software Command Line Tool is that the SDK environment is set to NOT build projects when it detects a change.

You have two choice here: You can manually build your projects when you need to, or you can enable the automatic build process. Use the instruction below to enable the automatic builds.

1-8. Enable or disable the automatic compilation capability.

1-8-1. Select **Project** > **Build Automatically**.

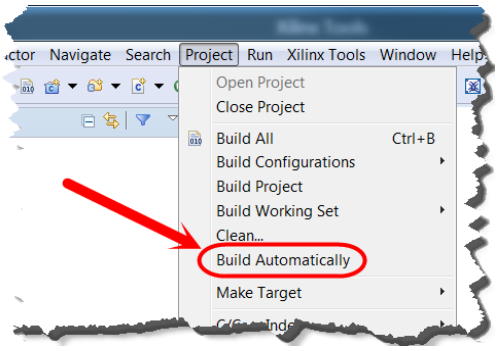


Figure 8-11: Selecting Build Automatically

This will toggle the Build Automatically option. When the Build Automatically option is active, a green check mark will appear next to it.

Finishing and Building the Software Application Step 2

The stopwatch application has been provided to you mostly finished. What remains is the enabling of the timer's interrupt and making the general interrupt controller (GIC) sensitive to this interrupt.

There are a number of functions (and macros) associated with setting up the interrupts. Let's quickly review some of the functions and macros that you will be using and why they are necessary.

Let's begin with looking at which functions associate with which aspects of the hardware. The figure below illustrates that there are two "places" that are associated with receiving interrupts. One is the GIC, which can take interrupts from a wide number of sources, including the peripherals in the IOP block, the PL, and SCU. The vector interrupt table in this device is substantial enough that each interrupt source can have its own handler associated with it. The other place sensitive to interrupts is the CPU itself. The CPU has relatively few interrupt inputs (IRQ and FIQ); however, it still has its own vector interrupt table.

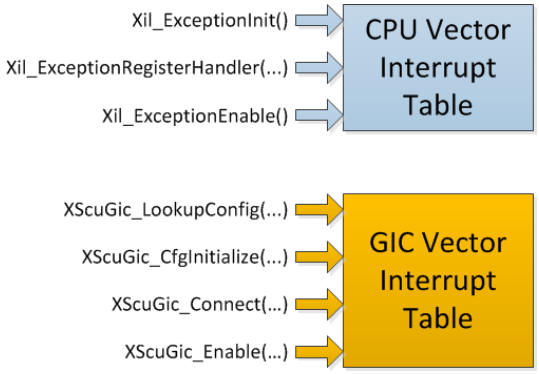


Figure 8-12: Interrupt Targets and Some Associated Function Calls (Zynq7 PS/GIC)

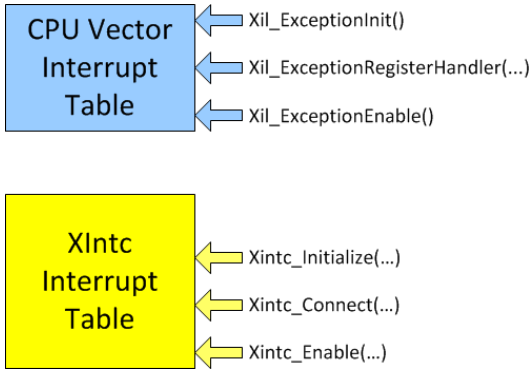


Figure 8-13: Interrupt Targets and Some Associated Function Calls (MicroBlaze Processor and INTC)

The functions are aptly named in that the ones associated with the GIC (used with the Zynq AP SoC) begin with "XScuGic_...", those associated with the Interrupt Controller (used with the MicroBlaze processor) begin with "Xintc_", and those associated with the CPU (in standalone for either processor) begin with "Xil_...".

Some of the functions can be called in any order; however, many of the calls must be made in a specific sequence in order to operate properly. The Zynq AP SoC GIC calls in order:

- XScuGic_LookupConfig – Looks up the device configuration based on the ID listed in *xparameters_ps.h*.

- XScuGic_CfgInitialize – Initializes the GIC. Clears the fields in the XScuGic structure, loads the vector table with sub handlers, and disables interrupt sources.
- XScuGic_Connect – Attaches an interrupt handler to a location in the interrupt vector table (IVT). Called once per used handler. If a device does not need a handler, then nothing needs to be done as the XScuGic_CfgInitialize places "stubs" in all the locations in the IVT.

The MicroBlaze processor XINTC calls in order:

- XIntc_Initialize – Initializes the INTC. Clears the fields in the XIntc structure, loads the vector table with sub handlers, and disables interrupt sources.
- XIntc_Connect – Attaches an interrupt handler to a location in the interrupt vector table (IVT). Called once per used handler. If a device does not need a handler, then nothing needs to be done as the XIntc_Initialize places "stubs" in all the locations in the IVT.

Next, the CPU's vector setup calls:

- Xil_ExceptionInit – Sets up a blank vector interrupt table. Must be called before registering any exception handlers or enabling any interrupts (standalone).
- Xil_ExceptionRegisterHandler – Registers the interrupt handler for the IRQ. In this case, it is the handler for the interrupt from the GIC.

For the Zynq AP SoC, the next block of code manages the setup for the SCU timer, which does not directly apply to how interrupts are implemented.

The following are the final stages of setting up the interrupts—enabling each interrupt from the timer through the GIC to the CPU. The specific order does not matter here as any break in this chain will cause the timer's interrupt to fail to reach the CPU.

- XScuTimer_EnableInterrupt (Zynq)/XIntc_Start (MicroBlaze processor) – Enable the timer's interrupt to the interrupt controller.
- XScuGic_Enable (Zynq)/XIntc_Enable (MicroBlaze processor) – Enable the interrupt from the interrupt controller to the CPU.
- Xil_ExceptionEnable – Enable the CPU to be sensitive to incoming interrupts.

While the above provides a brief outline of the order and basic behavior of the interrupts for this design, you may need more information...

2-1. You will now examine the Xilinx-provided documentation to reveal the necessary drivers to enable the GIC and make the necessary modifications to the existing code, save the file, and verify error-free compilation.

2-1-1. View the **system.mss** file in the edit window.

If the file is not already open, expand the **bsp** project in the Project Explorer tab, then double-click the **system.mss** file to open it.

2-1-2. Click the **Documentation** link next to *scugic* for the Zynq AP SoC or *intc* for the MicroBlaze processor.

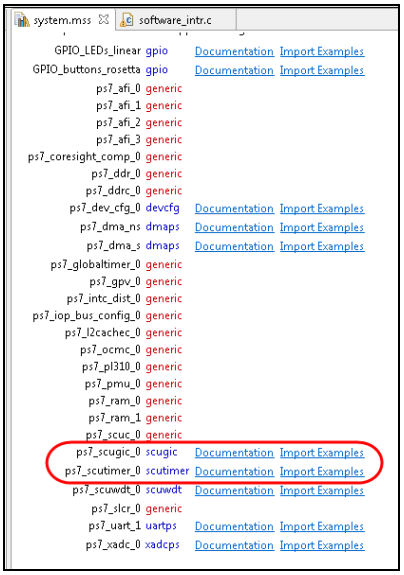


Figure 8-14: Opening the Documentation (scugic and scutimer)

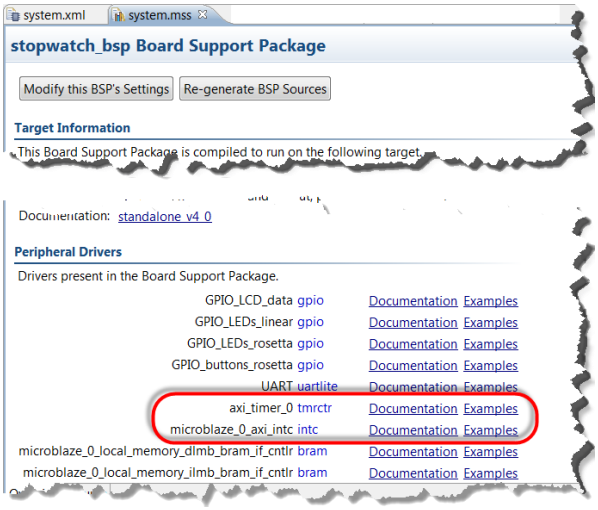


Figure 8-15: Opening the Documentation (axi_timer and axi_intc)

A browser window opens to the *Overview* tab for the peripheral.

2-2. View the scugic (Zynq AP SoC) or intc (MicroBlaze processor) API.

2-2-1. Click the **APIs** tab to see the alphabetized list of all of the drivers for the SCU GIC/intc peripheral.

2-2-2. Locate the entry for **XScuGic_Enable** (Zynq AP SoC) `or` **xintc.c** (MicroBlaze processor).

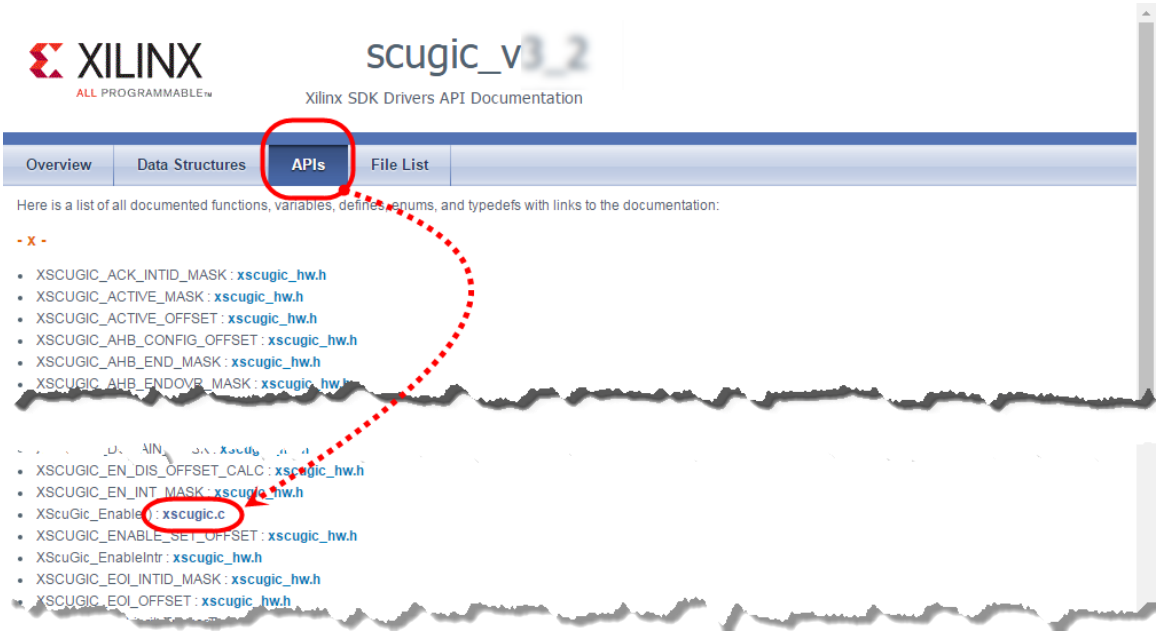


Figure 8-16: Locating the API for the SCUGIC

2-2-3. Click the **xscugic.c** link for the Zynq AP SoC or the **xintc.c** link for the MicroBlaze processor and view the function details.

The snippet below represents the Zynq AP SoC/xscugic.c entry. The MicroBlaze processor /intc.c entry is similar.

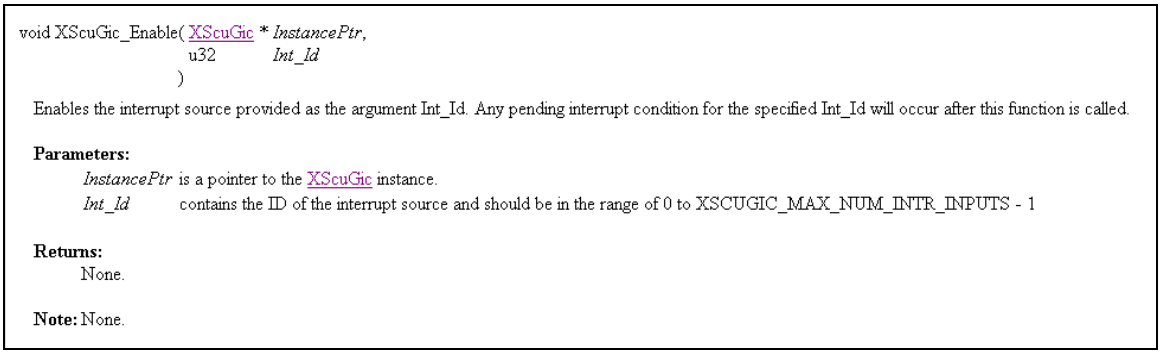


Figure 8-17: Viewing the XScuGic_Enable Function

Let's see how well you understand the interrupt structure and how to access the documentation.

Note: You could also have selected the File List tab and hyperlinked to xscugic.c. This means that you would have had to know that the driver you were looking for was in this file as well as having to search for the specific API name.

Question 1

Fill in the names of the function calls that you will make to access the items marked A through E in the figure below.

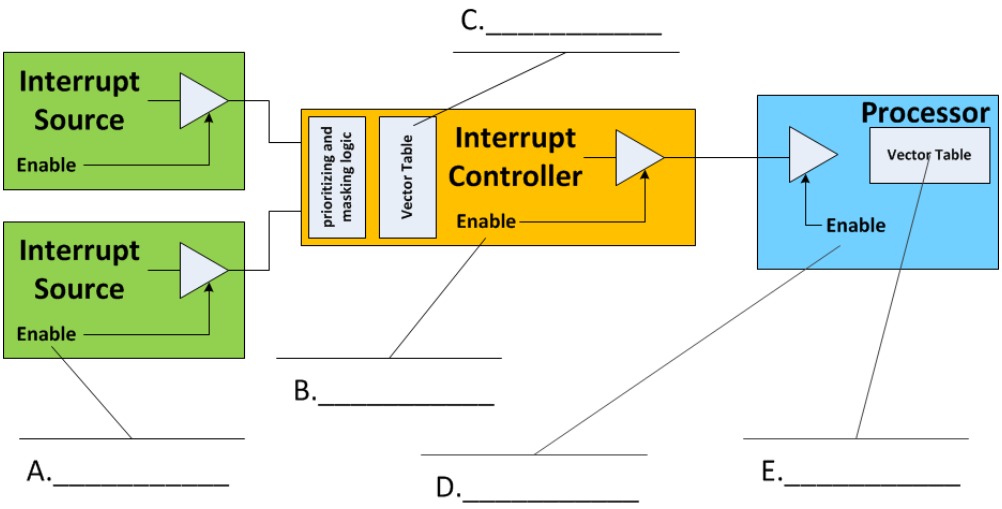


Figure 8-18: What Functions are Called for the Items Marked A-E?

2-3. Write the code to enable the interrupt and enable the timer interrupt.

2-3-1. Expand **SWinterrupt_app** > **src** to access the list of sources in the application project.

2-3-2. Double-click **stopwatch_intr.c** to open it in the editor.

2-3-3. [Optional] Enable line numbering.

If you do not recall how to enable line numbering, refer to the Lab Reference Guide > SDK > Enabling Line Numbering in the SDK Text Editor.

2-3-4. Locate the comment "*manage the SCU's GIC*" for the Zynq AP SoC or the "*manage the interrupt controller*" comment for the MicroBlaze processor.

If you do not recall how to perform a search, refer to the Lab Reference Guide > SDK > Finding Text in a Source File.

Here you will see that the snoop control unit (SCU), general interrupt controller (GIC), or interrupt controller (XIntc for the MicroBlaze processor) is initialized. The timer initialization and configuration are shown further down.

Question 2

What two important operations are missing from this code?

2-3-5. Test your skills and write the code to enable the interrupt from the GIC/XIntc to the Cortex-A9 processors or MicroBlaze processor.

Using the documentation shown above, make an effort to determine the proper function call. Add your code snippet after the line commented by:

[Zynq AP SoC users]: // Enable the SCU GIC interrupt here

[MicroBlaze processor users]: // Enable the interrupt controller here

If you get stuck, there is a code snippet at the end of the source code that solves this scenario:

Zynq AP SoC users: XScuGic_Enable(&intcInstance, TIMER_IRPT_INTR);

MicroBlaze processor users: XIntc_Enable(&intcInstance, TIMER_IRPT_INTR);

2-3-6. Continue to demonstrate your coding prowess by writing the one line of code to enable the timer interrupt.

Using the documentation shown above, make an effort to determine the proper function call. Add your code snippet after the line commented by:

[Zynq AP SoC users]: // Enable the timer interrupt here

[MicroBlaze processor users]: // Enable the INTC interrupt here (from the INTC to the processor)

If you get stuck, there is a code snippet at the end of the file that solves this scenario:

Zynq AP SoC users: XScuTimer_EnableInterrupt(&timerInstance);

MicroBlaze processor users: XTmrCtr_EnableIntr(TIMER_BASE_ADDR, TMR0);

2-3-7. Save the **stopwatch_intr.c** file.

Question 3

Where can BaseAddress(es), the constants relating to the DEVICE_IDs of various peripherals be found?

Question 4

What level of driver service is this? How can you tell? What file would be referenced to understand its operation?

Configuring the Device and Testing the Application Step 3

This final step will have you configure the Zynq AP SoC/MicroBlaze processor, download the application, and test the program for proper stopwatch operation. The hardware portion of the design has already been implemented for you. This lab assumes that the hardware board and download cabling are in place.

3-1. Program the device.

The bitstream resides within the SDK workspace as it was imported when the hardware platform was created.

3-1-1. Select **Xilinx Tools** > **Program FPGA** or click the icon.

The Program FPGA dialog box opens.

The default bitstream is the bitstream that is part of the collection of files that was exported from the Vivado Design Suite.

The BMM/MMI file is used to describe how block RAM memory clusters are loaded with instruction and/or data information that is contained as part of the bitstream. The BMM format is used in older versions of the tool, while the MMI format is used in newer versions of the tool. Typically, Zynq All Programmable SoC-only designs do not require a BMM/MMI file, and MicroBlaze processor designs do require their use.

ELF files, under the Software Configuration section of the dialog box, are also related to BMM/MMI usage. If no BMM/MMI file is specified, then this section will remain empty.

3-1-2. Keep all default settings and click **Program** to program the programmable logic portion of the device.

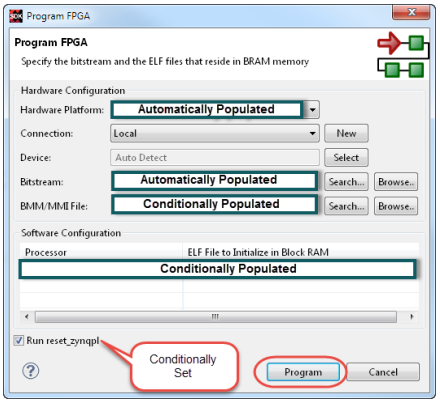


Figure 8-19: Programming the FPGA

It will take about a minute to configure the programmable logic. A progress bar will appear to show how far along the programming process is. Typically, the progress bar will pause near the halfway mark for a few seconds before completing the configuration. The result of the FPGA configuration can be viewed in the SDK Log tab at the bottom of the IDE.

The SDK terminal is an interface that only supports serial port/UART communications. The more general Terminal tab is able to support other formats, such as SSH and Telnet.

3-2. Locate the SDK Terminal tab. Generally this tab is found in the same panel as the console.

3-2-1. Select **Window** > **Show View** > **Other** > **Xilinx** > **SDK Terminal** to open the SDK Terminal tab if it is not currently visible.

This tab typically opens in the same window as the console.

3-3. Configure the SDK Terminal.

3-3-1. Click the green '+' sign to open the Connect to Serial Port dialog box.

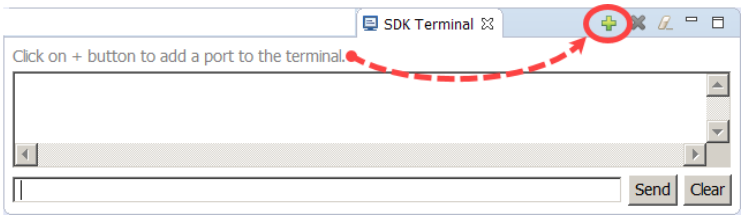


Figure 8-20: Adding/Associating a Port to the Terminal

A pop-up appears asking you to configure the settings for the serial port.

3-3-2. Select the serial port that is connected to the device you want to communicate with (1).

This is the port number associated with the Serial Port/USB connection from your board. This is often the highest-numbered com port, but not always. Your board must be powered on in order to see this port.

3-3-3. Set the baud rate to **115200** (2).

3-3-4. Leave the other settings at their defaults.

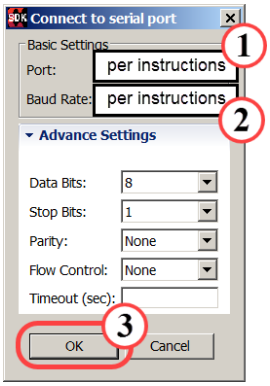


Figure 8-21: Configuring the SDK Terminal

3-3-5. Click **OK** to save these settings and begin the terminal session (3).

3-4. Run *SWinterrupt_app*.

3-4-1. Right-click **SWinterrupt_app**.

3-4-2. Select **Run As** > **Launch on Hardware (System Debugger)**.

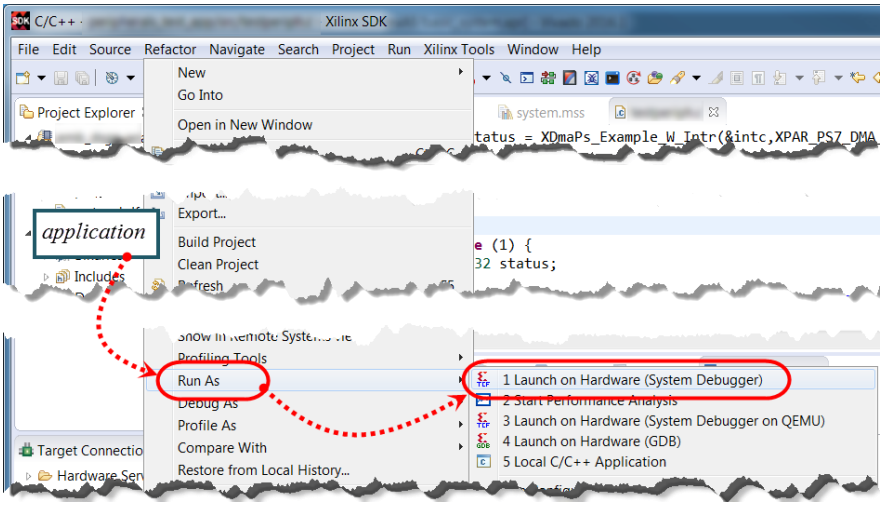


Figure 8-22: Launching a Run on Hardware

3-4-3. Click **Yes** if you are asked to terminate a previous run.

The program launches on the hardware.

3-5. Verify proper stopwatch operation.

Time is displayed in two ways. The LEDs on the board display seconds and tenths of seconds as: upper nibble represents seconds in BCD, the lower nibble represents tenths of seconds in BCD. The time is also reported via the serial port.

The button functions are:

- **START → BTN1 (West)**
- **STOP → BTN2 (North)**
- **RESET → BTN3 (East)**

3-5-1. Press the START (West) button and verify that the time value is incrementing.

3-5-2. Press the STOP (North) button and verify that the time value stops incrementing.

3-5-3. Press the RESET (East) button and verify that the time value resets to zero.

3-5-4. Repeat in any combination. Does the stopwatch behave as expected?

3-6. Exit the SDK tool and power off the board.

3-6-1. Select **File** > **Exit** to close the SDK tool.

3-6-2. Power off the development board.

Summary

You just finished examining an interrupt-driven stopwatch application where a timer generated a 10ms "heartbeat". For each beat of the timer, an interrupt was generated and the interrupt handler was called.

You wrote code to enable the interrupt out of the hardware timer to the GIC/interrupt controller and enabled the output of the GIC/interrupt controller to the processors.

You also leveraged the documentation on *scu_gic* and *scu_timer* to locate the necessary information to complete this task. Finally, you verified the operation of the code on hardware.

Answers

1. Fill in the names of the function calls that you will make to access the items marked A through E in the figure below.

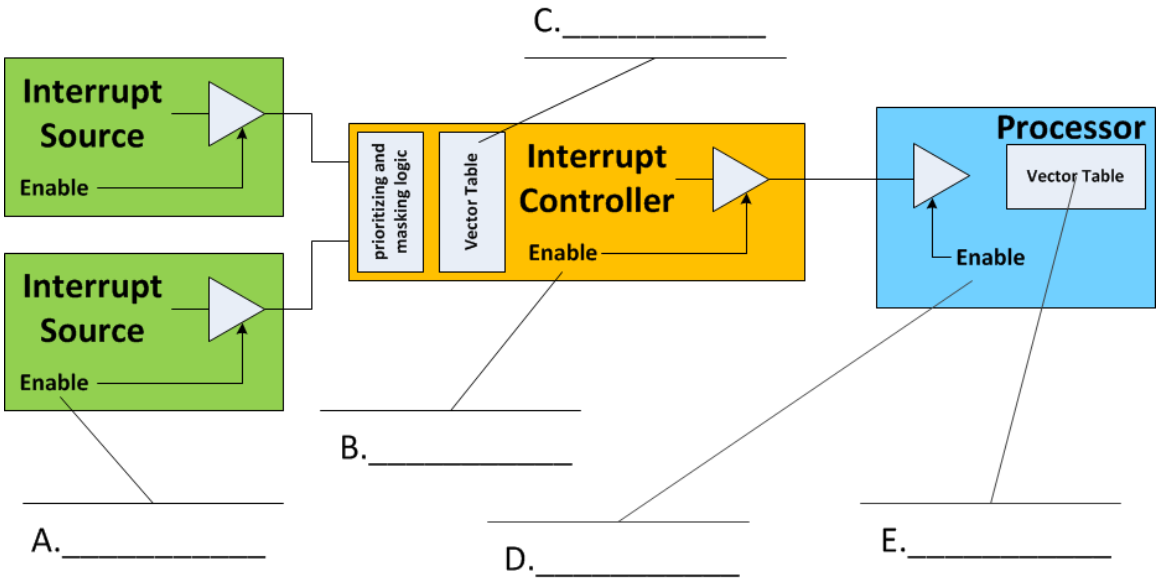


Figure 8-23: What Functions are Called for the Items Marked A-E?

Answer:

A:

The *interrupt source* could be anything from a button to an internal peripheral responding to external stimulus (think UART or EMAC) to an internal peripheral generating its own interrupt (think timer, random interval generator, etc.).

When a timer is used, as in this design, the timer is configured to generate an interrupt based on some preloaded value. When the timer reaches the designated value, an interrupt is generated. The interrupt needs to be made available external to the timer peripheral and the function to do this is dependent on the peripheral itself.

The Zynq AP SoC contains a number of timers and the one used in the Zynq AP SoC version of this lab is the SCU timer. The required function call is *XScuTimer_EnableInterrupt()*. The AXI_Timer peripheral used in the MicroBlaze processor design uses a macro to enable the interrupt, *XTmrCtr_EnableIntr()*.

B:

The interrupt controller is a powerful and capable device that can dynamically mask incoming interrupt sources and enable or suppress its own interrupt generation to the next higher level of interrupt controller or CPU.

For the Zynq AP SoC, the function used to enable any interrupt to the processor is *XScuGic_Enable()*. The MicroBlaze processor equivalent is *XIntc_Enable()*.

C:

The vector table maps specific handlers to each interrupt. Typically there is one call per interrupt source.

The GIC in the Zynq AP SoC uses *XScuGic_Connect()* to map the handler to the proper entry in the vector table.

A nearly identical call is available for the INTC device used with the MicroBlaze processor: *XIntc_Connect()*.

D:

The processor can decide to accept interrupts from the interrupt controller (or directly from an interrupt source) or not when *Xil_ExceptionEnable()* is called.

Note that this is the same for both processors when running under the standalone platform.

E:

Like the interrupt controller's vector table, the processor has its own interrupt vector table.

The primary difference between the interrupt controller's table and the processor's table is the interrupt controller is concerned only with interrupts emanating from the *outside* world while the processor must be concerned with software-generated interrupts and exceptions, such as what to do with a divide by zero condition.

The function *Xil_ExceptionRegisterHandler()* adds the appropriate handler to deal with interrupts coming from the interrupt controller.

2. What two important operations are missing from this code?

   First, the interrupt (often referred to an an exception) handler must be registered with the GIC so that the proper code gets called when this event occurs. Second, the timer's interrupt to the GIC must be enabled.

3. Where can BaseAddress(es), the constants relating to the DEVICE_IDs of various peripherals be found?

   All Zynq All Programmable SoC hardware-related constants can be found in one of two header files: *xparameters_ps.h* (which includes information regarding the peripherals included in the PS and *xparameters.h* (which includes information about the portion of the embedded system located in the PL).

   MicroBlaze processor-only designs in non-Zynq All Programmable SoC systems will only have the *xparameters.h* file.

4. What level of driver service is this? How can you tell? What file would be referenced to understand its operation?

   This is a high-level driver. All Level 0 drivers are defined in the **_l.h* files.

Lab 9: Linux Application Development

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

This lab explains how to develop a Linux application to access the general-purpose input/output (GPIO) that is connected to the hardware resources available on the development board. A C-based source file that contains the GPIO access information is provided. This source file is intentionally left incomplete and requires the code to read the DIP switch value. You will complete the code as part of this lab.

Objectives

After completing this lab, you will be able to:

- Create an SDK software application workspace
- Create and build a Linux software application project
- Download and run an application on development hardware

Introduction

The Zynq®-7000 All Programmable SoC Linux solution combines the benefits of the Linux operating system together with PetaLinux and the Xilinx SDK tools towards designing software applications on this platform. This lab demonstrates how to access GPIO in a Linux application.

There are two methods for interacting with GPIO from user space:

- `sysfs` interface
- Linux kernel drivers

The `sysfs` interface is a very simple way to access the GPIO from user space and has the advantage of requiring very little setup. This option is a good choice for manually checking the status of an input or writing values to outputs; this method should not be used where interrupts are required.

There are some important points with `sysfs`:

1. The GPIO controllers are visible in `/sys/class/gpio`. Each controller controls a number of GPIO signals.
2. The GPIO signals must be exported into `sysfs` before they can be manipulated. The number of the GPIO signal must be written to the GPIO export file to cause this to happen.

3. After the GPIO signal is exported, a new directory based on the GPIO signal number will appear in `/sys/class/gpio`. Under this directory are `direction` and `value` files that can be read and written.

You will find the proper gpio, export the gpio, and access the `direction` and `value` files.

For more information, go to www.wiki.xilinx.com/Linux+GPIO+Driver.

Based on the Linux image usage, the table below explains the label for the peripherals (LEDs and Switch).

Linux Image Use	LEDs (Base Board)	Switch (Base Board)
SDcard_zc702_base (ZC702 board)	DS15 to DS22	SW12 (2)
SDcard_zed_base (ZedBoard)	LD0 to LD7	SW0

For the ZC702 board, the file accesses the GPIO, which is connected to the DIP switch on the ZC702 board SW12[2] and prints the value of the switch whenever there is a change in it.

For the ZedBoard, the file contains the GPIO, which is connected to the DIP switch (SW0) on the ZedBoard and prints the value of the DIP switch whenever there is a change in it.

For reference the GPIO IDs (to be used in the C code) are shown in the table below.

Linux Image Use	LEDs – GPIO ID (Base Board)	Switch – GPIO ID (Base Board)
SDcard_zc702_base (ZC702 board)	898	897
SDcard_zed_base (ZedBoard)	898	897

General Flow

Creating an SDK Tool Workspace Step 1

The first step in this lab is to create an SDK tool workspace.

The working directory of the SDK workspace for this lab is *C:\training\Linux_App_Dev\lab*. You will create the SDK workspace and software platform for the hardware in this directory.

Once SDK is launched and a workspace is set up in this directory, you will not be able to move these directories.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\Linux_App_Dev\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\Linux_App_Dev\lab** directory and click **OK**.

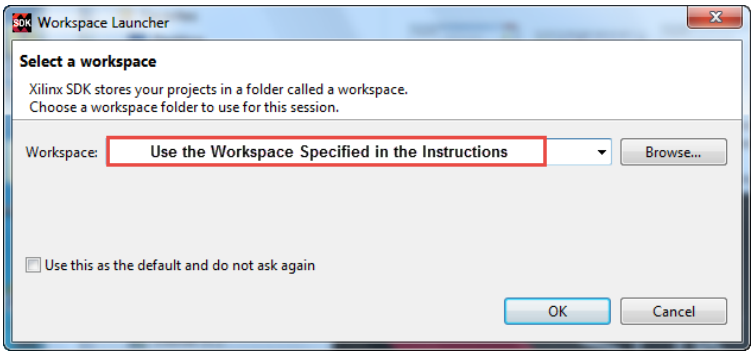


Figure 9-1: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

Adding a Software Application Step 2

Using the Application Project Wizard is a quick way to set up a Linux C or C++ software application project that targets an existing processor and Linux version. Based on the dialog box choices, the appropriate toolchain is selected for pre-processing, compiling, assembling, and linking.

2-1. Create a new C/C++ application project named *appDev*.

2-1-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

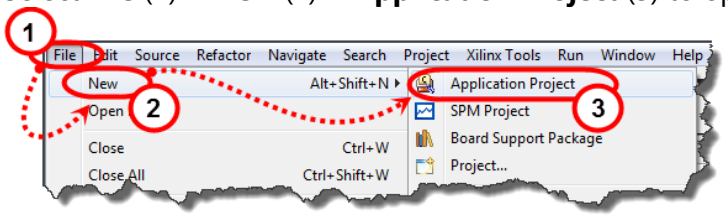


Figure 9-2: Creating an Application Project

2-1-2. Enter **appDev** as the project name (1).

2-1-3. Ensure that **linux** is selected from the OS Platform drop-down list (2 and 3).

2-1-4. Ensure that **ps7_cortexa9** is selected from the Processor drop-down list.

2-1-5. Select your preferred language: C or C++ (4).

This selects which tools will be used to compile your code.

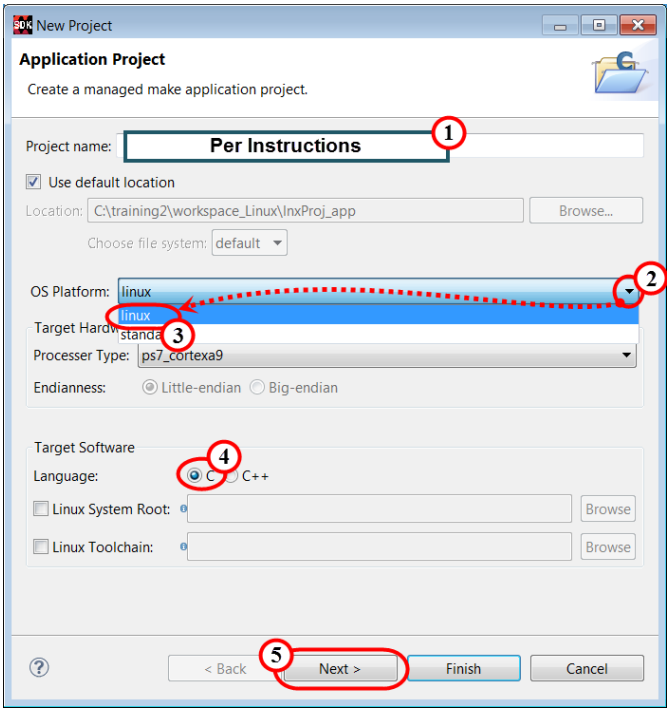


Figure 9-3: Selecting Linux for the New Application Project

2-1-6. Click **Next** to view the templates available for Linux applications (5).

2-1-7. Select **Linux Empty Application** (1).

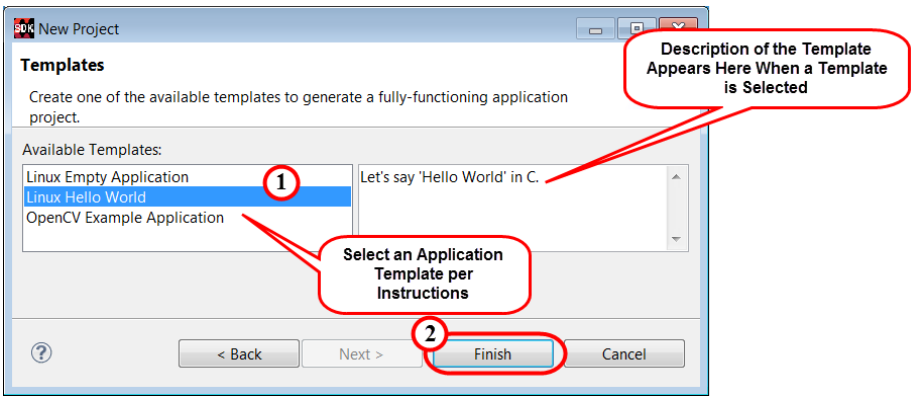


Figure 9-4: Selecting a Linux Template

2-1-8. Click **Finish** to close the dialog box and create the new project (2).

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

2-2. Add *linux_appDev.c* to the application.

The preferred method for importing sources is shown here.

2-2-1. Expand the project named **appDev** > **src** using the Project Explorer.

2-2-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

2-2-3. Select **Import** to open the Import Wizard (2).

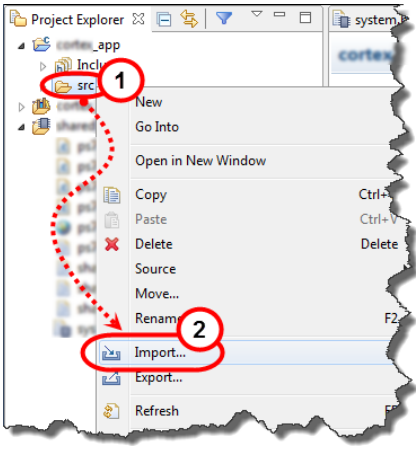


Figure 9-5: Importing a Resource File

The Import Wizard dialog box opens.

2-2-4. Expand **General** (1).

2-2-5. Select **File System** as you will be selecting individual files directly from the file system (2).

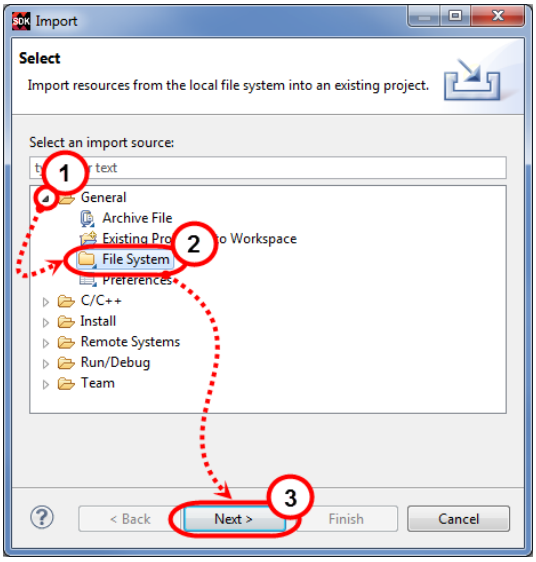


Figure 9-6: Selecting File System

2-2-6. Click **Next** to advance to specifying the files to import (3).

2-2-7. Browse to *C:\training\Linux_App_Dev\support* in the *From directory* field.

2-2-8. Select the file(s) by checking the box beside **linux_appDev.c**.

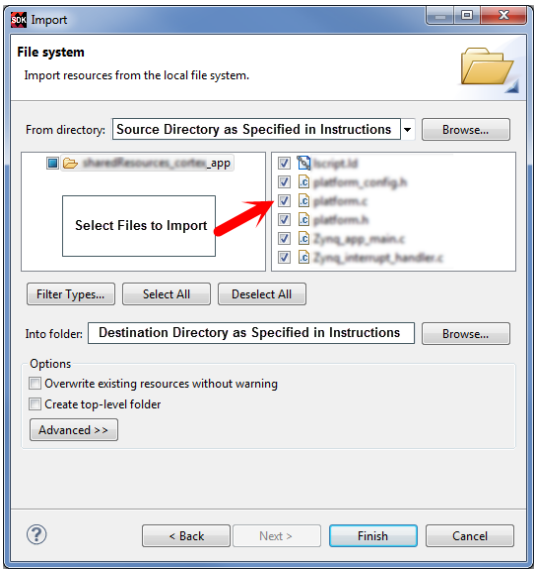


Figure 9-7: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

2-2-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

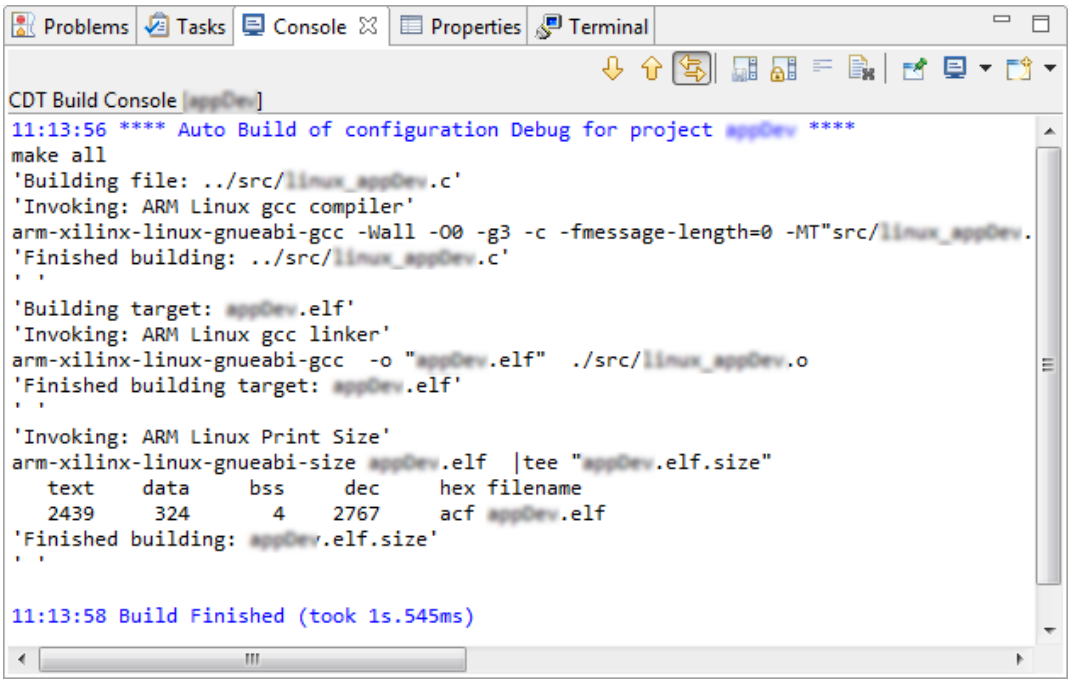


Figure 9-8: Successful Software Application Build

2-3. Open the source file that you imported in the previous step.

2-3-1. Expand the **appDev** project in the Project Explorer tab.

2-3-2. Expand the **src** folder.

2-3-3. Double-click **linux_appDev.c** to open it in the edit window.

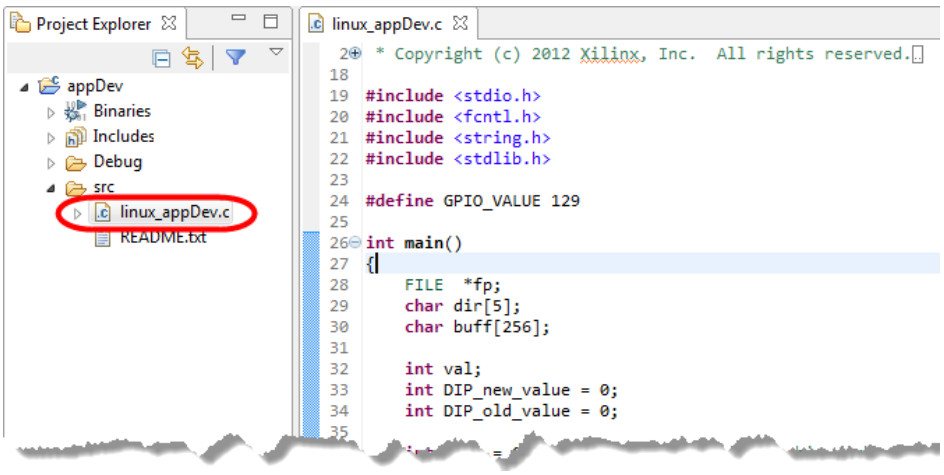


Figure 9-9: Opening the linux_appDev.c File

Downloading and Running the Program Step 3

Superior performance and additional capabilities are available when using an Ethernet cable to communicate with the target platform rather than using a serial download cable. When an Ethernet connection is used, the PC host requires changing to a static IP address. This procedure may have already been performed when the laptop computers were set up. Check with your instructor to determine if the following step can be skipped.

3-1. Change the IP address of the host laptop to a static IP address.

If you do not recall how to change the IP address of the host laptop from dynamic to static, refer to the "Setting Static Host IP Address on Laptop (Windows 7)" topic under the Software Requirements section in the *Lab Reference Guide*.

3-1-1. Verify with the instructor that the host PC (laptop) has a static IP address set set to **192.168.1.11**.

3-2. Verify that the SD card with the Linux image is present.

3-2-1. Make sure that the SD card with a Linux image is present on the ZC702 or ZedBoard.

For the ZC702 board, copy the files from the *C:\training\Linux_App_Dev\support\SDcard_zc702* archive to the root of the SD card.

For the ZedBoard, copy the files from the *C:\training\Linux_App_Dev\support\SDcard_zed* archive to the root of the SD card.

The ZC702 or ZedBoard evaluation hardware platform has jumpers (or switches) to select the boot mode for the processor.

3-3. Verify that the ZC702 or ZedBoard hardware platform is properly connected and the jumpers (or switches) are configured to boot from the SD card.

3-3-1. Confirm that the power to the evaluation board is OFF.

3-3-2. Ensure that the connections between the host PC and the hardware evaluation board are correct (ask for assistance from the instructor if necessary).

The connections include power, serial bridge USB, Ethernet, and the USB Platform download cables.

3-3-3. Make sure that the SD card with a Linux image is inserted into the board card slot.

3-3-4. Verify the jumper settings on the board are set up to boot from the SD card.

3-3-5. Refer to the "Jumper/Switch Settings ZC702 or ZedBoard - Boot from SD Card " topic under the Hardware Requirements - ZC702 or ZedBoard Hardware Setup section in the *Lab Reference Guide*, or ask your instructor for assistance.

3-4. Apply power to (turn on) the ZC702 or ZedBoard hardware platform.

3-4-1. Make sure that AC power is connected to the power brick.

3-4-2. Slide the power switch to the on position.

Some LEDs on the board will illuminate when the board is powered.

The serial UART terminal provides a simple and straightforward way to collect and transmit serial information using the RS-232 protocols. Most modern PCs lack the traditional DB-9 connectors for RS-232 communication and instead use the USB ports. From the PC's perspective there is still a COMx port being used; however, the actual COM port number is not known until USB enumeration. The PC is capable of supporting multiple COM ports, so it is important to know which connection is the one to use when communicating with the development board.

When using a PC COM port with communications software (terminal emulator) on the PC such as the SDK terminal tab, Tera Term, or other software, it is necessary to identify the COM port number associated with serial connection to the evaluation board. This is done by identifying the USB COM port driver and which COM port is being assigned.

3-5. Determine which COM to use to access the USB serial port on the ZC702 or ZedBoard.

3-5-1. Make sure that the development board is powered on and the serial UART device USB cable is in place.

This ensures that the USB-to-serial bridge will be enumerated by the PC host.

3-5-2. Open your computer's **Control Panel**.

Note that the Start button is typically located in the lower left corner of the screen. Occasionally, as shown below, it is in the upper left corner.

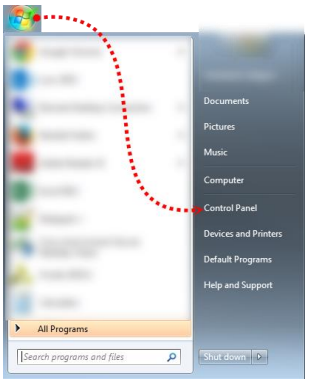


Figure 9-10: Opening the Control Panel from Windows 7

3-5-3. Click **Hardware and Sound** > **Device Manager** to open the Device Manager window.

Note: You may be asked to confirm opening the Device Manager. If so, click **Yes**.

3-5-4. Expand **Ports (COM & LPT)**.

3-5-5. **ZC702 board users:** Locate **Silicon Labs CP210x USB to UART Bridge (COM#)**.

ZedBoard users: Locate **USB/Serial Cypress (COM#)**.

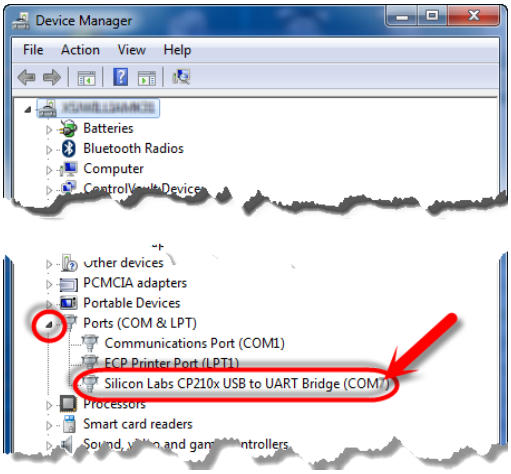


Figure 9-11: Locating the Silicon Labs Com Port Driver

The **#** indicates the port number for this serial connection.

Known issue: The present version of the SDK tools contains an incompatibility between the USB/Serial Cypress host drivers used for the ZedBoard and the SDK terminal program. While the SDK terminal window is the preferred console program, the Tera Term console will be used for the ZedBoard. The ZC702 board works fine with the SDK terminal. The following instructions will branch, depending on which evaluation board is used; outputs in either terminal should be identical.

3-5-6. Close the Device Manager by clicking the red 'X' in the upper right corner of the window.

3-6. ZedBoard users: Skip to the next step.

ZC702 board users: Open the SDK terminal program when using the ZC702 board.

3-6-1. Select the **Terminal** tab to access the terminal icons.

If the terminal tab is not visible, select **Window** > **Show View** > **Terminal.** Note that more than one terminal can be enabled at a time.

3-6-2. Click the **Settings** Icon ( ).

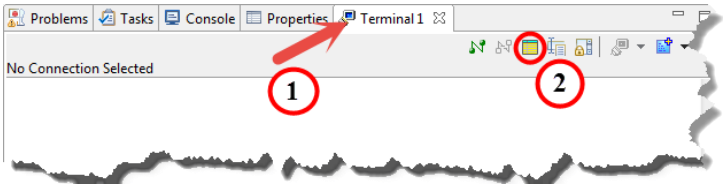


Figure 9-12: Accessing the Terminal Settings

Alternatively, you can also click the **Connect** icon ( ) to open the Terminal Settings dialog box. If the terminal was previously configured, this will open it with those settings and connect to the associated COM port.

3-6-3. Configure the settings as shown in the following figure.

- Select the connection type as **Serial**
- Select the port as the COM **#** discovered in the last step
- Set the baud rate to **115200**

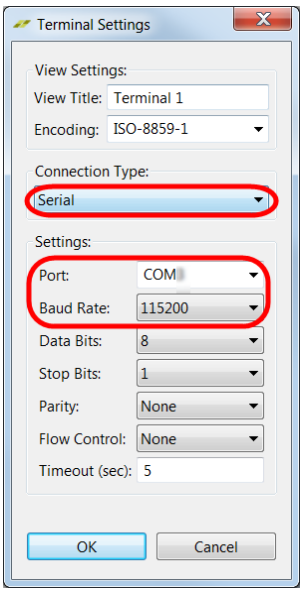


Figure 9-13: Configuring the Terminal Settings

3-6-4. Click **OK.**

The terminal session will be connected to the associated COM port on the PC.

3-7. ZC702 board users: You have already associated a serial terminal with your board. Skip to the next step.

ZedBoard users: Open the Tera Term terminal program.

3-7-1. Double-click the **Tera Term** icon from the Windows desktop to launch Tera Term.

Alternatively, you select **Start** > **All Programs** > **Tera Term** > **Tera Term**.

3-7-2. Select **File** > **New Connection**.

3-7-3. Select **Serial** as the connection (1).

3-7-4. Click the **Port** pull-down menu to view the available COM ports.

Note: If your port is not listed, exit Tera Term, power cycle your board and re-start this step.

3-7-5. Select the COM **#** discovered in the last step (3).

Note that the ZC702 will show the Silicon Labs driver as shown in the figure below and the ZedBoard will show the Cypress driver.

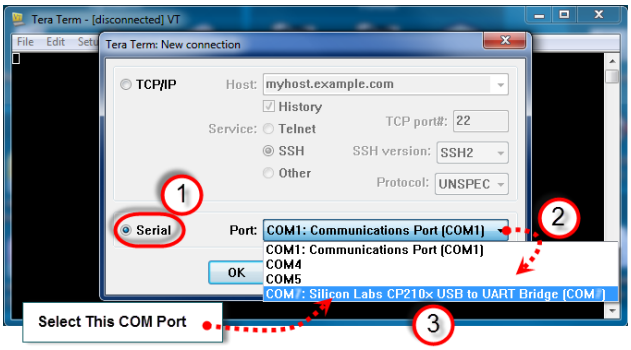


Figure 9-14: Selecting the COM Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

3-7-6. Click **OK**.

The terminal console window opens.

3-7-7. Select **Setup** > **Serial Port**.

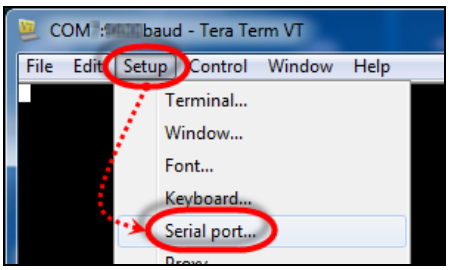


Figure 9-15: Opening the Tera Term Serial Port Setup Window

The Tera Term Serial Port Setup dialog box opens.

3-7-8. Confirm that the proper serial port has been selected (1).

3-7-9. Set the baud rate to **115200** (2).

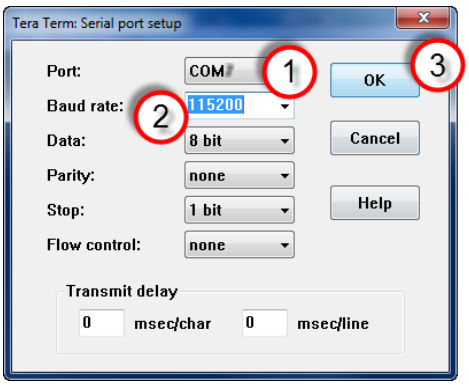


Figure 9-16: Setting the Parameters for the Serial Port

Note: The COM port setting is specific to the computer being used and may need to be different than shown. Use the COM port **#** that was discovered in the previous step.

3-7-10. Click **OK** (3).

Tera Term is now configured to receive and transmit serial information to/from the evaluation board.

Linux booting will be displayed in the Terminal window.

3-7-11. Wait until booting has completed.

3-8. Set the board IP address.

3-8-1. Enter `root` as the ESD_Linux_Hw login and password in the SDK terminal (ZC702 board) or Tera Term window (ZedBoard).

3-8-2. Enter the following command to set the IP address:

```
ifconfig eth0 192.168.1.10
```

Note: Find the network name (e.g., `eth0`) via the `ifconfig` command.

3-8-3. Enter the following command to verify the IP address:

```
ifconfig eth0
```

The response should be similar to the following:

```
eth0    Link encap:Ethernet  HWaddr 00:0A:35:00:01:22
        inet addr:192.168.1.10  Bcast:192.168.1.255
Mask:255.255.255.0
        inet6 addr: fe80::20a:35ff:fe00:122/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:23 errors:0 dropped:0 overruns:0 frame:0
        TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:2913 (2.8 KiB)  TX bytes:1446 (1.4 KiB)
        Interrupt:54 Base address:0xb000
```

3-8-4. Ping the host in the Linux terminal console to verify connectivity between the host and the target:

```
ping 192.168.1.11 -c 1
```

If the ping was successful, you can continue to the next section.

If it was not successful, follow the procedure below to disable the Windows firewall.

3-8-5. Disable the Windows firewall via the the Control Panel (**Start** > **Control Panel** > **System and Security** > **Windows Firewall**).

3-8-6. Select **Turn Windows Firewall on or off** from the options in the left sidebar and then select the **Turn off Windows Firewall** option for both network location settings.

3-9. Review the source code and open the Linux console.

3-9-1. Review the **linux_appDev.c** file.

3-9-2. Open the SDK Terminal window (ZC702 board users) or Tera Term application (ZedBoard users) to gain access to the Linux console.

The following instruction is to find the address of the GPIO connected to the DIP switch. There may be multiple GPIOs in the system and they will need to be inspected to find their base address and determine if this is the correct base address you are looking for. The base address will be found by viewing the label file available under `/sys/class/gpiochip<ID>/label`.

3-10. Find the <ID> for the GPIO that has a base address of `0x41210000`.

3-10-1. Use the following command to see the available GPIOs in the system:

```
# ls /sys/class/gpio
```

An example output is shown in the image below.

```
root@ESD_Linux_Hw:~# ls /sys/class/gpio
export   gpiochip129  gpiochip130  gpiochip138  unexport
```

There are three gpiochip<ID>s. gpiochip129, gpiochip130 and gpiochip138.

3-10-2. Enter the command below to find the base address of the GPIO with and ID of <ID>.

```
# cat /sys/class/gpio/gpiochip<ID>/label
```

In the output from the command, there will be an address listed.

3-10-3. Continue entering the command for the different gpiochips until you find the one that has an address of `0x41210000`

3-10-4. When you find the correct gpiochip<ID> entry, take note of the <ID> number.

3-11. Open the SDK program and edit the source file.

3-11-1. Open the SDK program.

3-11-2. Find **GPIO_VALUE** in the *linux_appDev.c* file (near line number 24).

3-11-3. Update **GPIO_VALUE** with the <ID> just obtained if needed.

3-11-4. Locate the comment "Check the direction of the GPIO - switch" (near line number 50).

You can now review the code on how the direction of the GPIO has been read.

3-11-5. Locate the comment "Write a code to open the necessary file to read the DIP Button file" (near line number 71).

3-11-6. Insert the following line of code below the comment:

```
sprintf(buff, "/sys/class/gpio/gpio%d/value", gpio);
```

3-12. Find the LED peripheral GPIO <ID> and update the code to blink when there is a change in the DIP switch.

3-12-1. Follow the same steps explained previously to find the LED peripheral GPIO <ID>.

The base address for the LEDs peripheral is `0x41200000`.

After finding the GPIO ID, follow the step below.

3-12-2. Locate the comment "For example - gpio-demo -g 130 -o 255" (near line number 110 in the image).

3-12-3. Replace 130 with the actual ID in line numbers 114 and 116 in the source file if the GPIO <ID> is different than 130.

```
108             while (count--){
109
110                 // gpio-demo -g GPIO_BASE -o value
111                 // For example - gpio-demo -g 130 -o 255
112                 system("gpio-demo -g 130 -o 255");
113
114                 system("gpio-demo -g 130 -o 0");
115             }
116         }
117         fclose(fp);
118
119     }
```

Figure 9-17: Updating the GPIO ID – LEDs Peripheral

3-12-4. Save the file and make sure that the file compiles without any errors.

3-13. Create a Run configuration. Run configurations associate an ELF object file to a target for execution. In this case, the target is a hardware board accessed over a network TCP/IP connection.

3-13-1. Click the **C/C++** perspective (top right) to return to the original perspective.

3-13-2. Right-click the **appDev** project in the Project Explorer window.

3-13-3. Select **Run As** > **Run Configurations**.

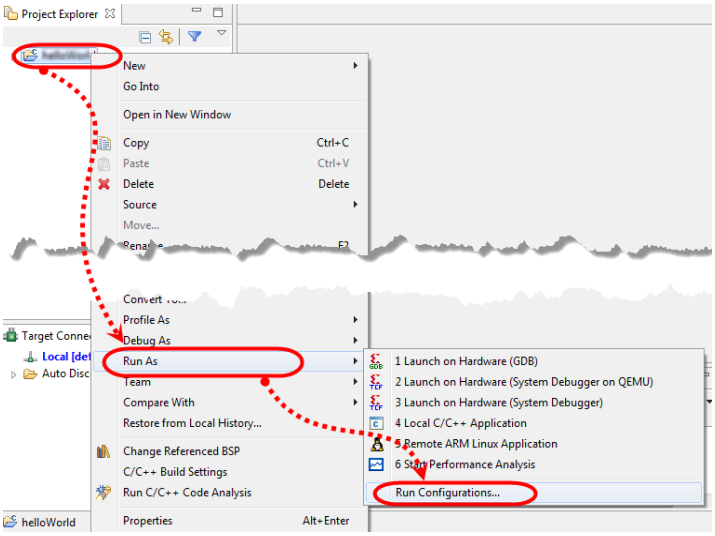


Figure 9-18: Selecting Run Configurations

3-14. Configure the debug type and host connection.

3-14-1. Double-click **Xilinx C/C++ application (System Debugger)** to create a launch configuration.

3-14-2. Select **Linux Application Debug** from the Debug Type drop-down list.

3-14-3. Click **New** next to the Connection drop-down list.

3-14-4. Enter **ZynqBoard** in the Target Name field.

3-14-5. Enter **192.168.1.10** in the Host field.

3-14-6. Enter **1534** in the Port field.

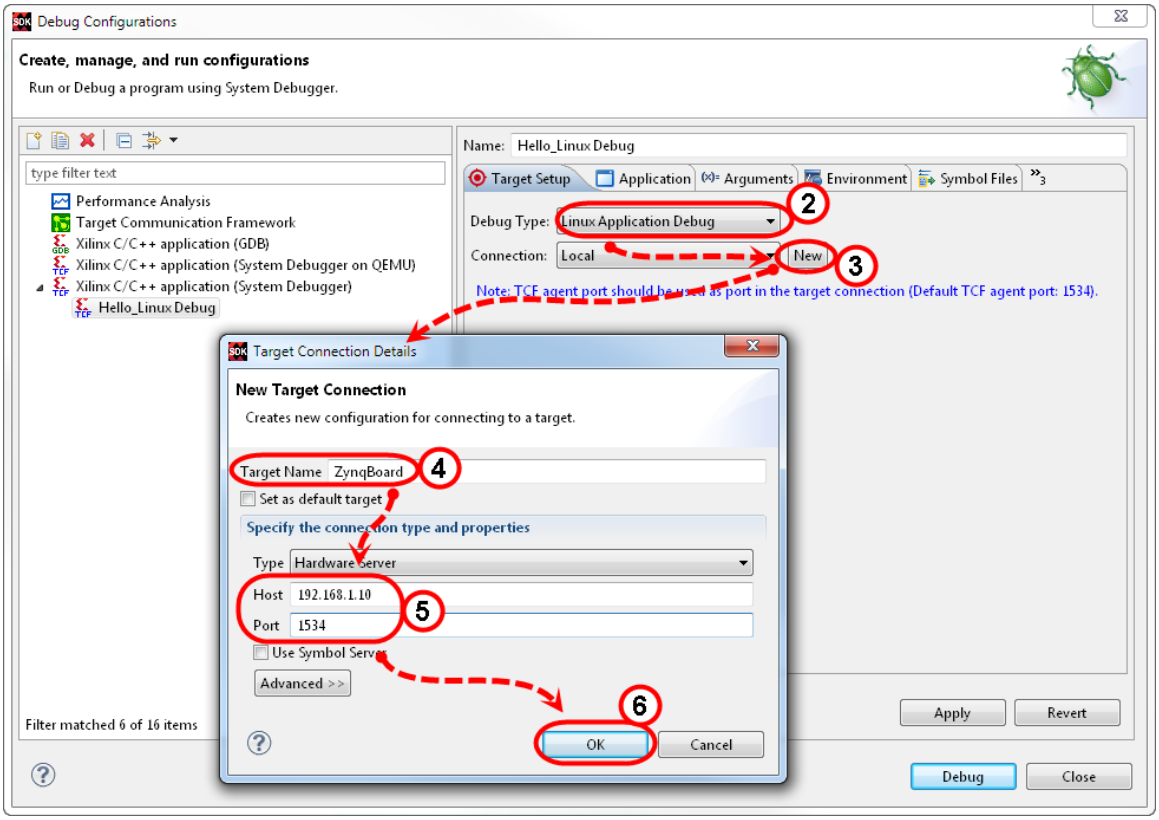


Figure 9-19: Selecting the Debug Type and Connection

3-14-7. Click **OK**.

3-15. Select the application and remote file path to copy the ELF file to the board.

3-15-1. Select the **Application** tab.

3-15-2. Click **Browse** next to the Local File Path field.

3-15-3. Select the local file path to be *C:\training\Linux_App_Dev\lab\appDev\Debug\appDev.elf*.

3-15-4. Enter **/tmp/appDev.elf** in the Remote File Path field.

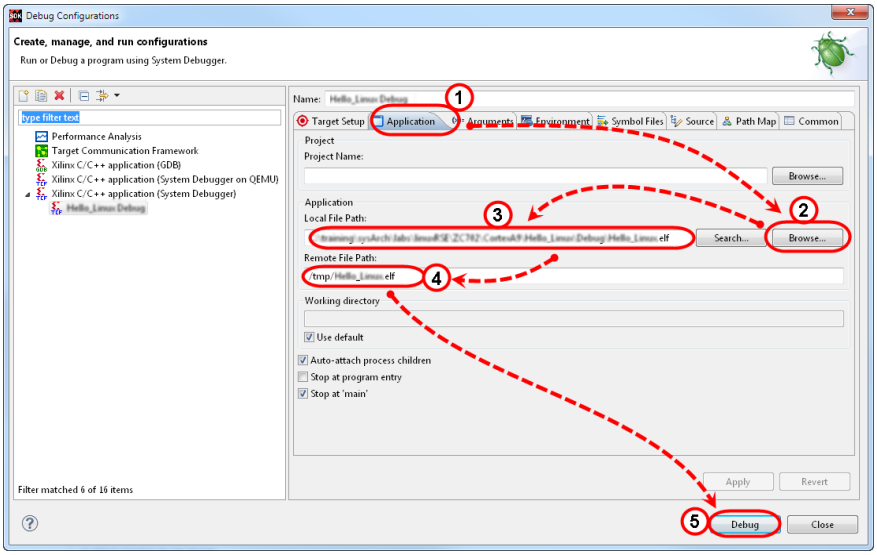


Figure 9-20: Selecting the Local File Path and Remote File Path

3-15-5. Click **Run**.

The output will be displayed in the Console window.

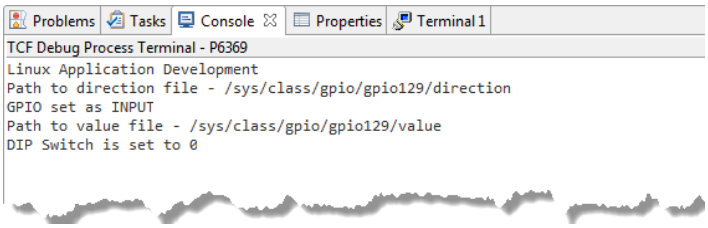


Figure 9-21: Viewing the Output

3-15-6. Change the switch position and view the output in the Console tab.

Note: Whenever there is a change in the switch value, the LEDs blink for few seconds.

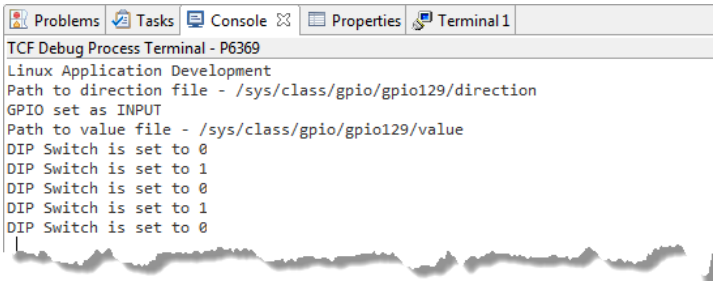


Figure 9-22: Viewing the Output after the Change in the Switch Position

3-16. Exit the SDK tool and power off the board.

3-16-1. Select **File** > **Exit** to close the SDK tool.

3-16-2. Power off the development board.

Summary

You created an application to read the DIP switch value on the ZC702 or ZedBoard and blink the LEDs whenever the DIP switch value changed.

Answers

Since there were no questions in this lab, this section is intentionally left blank.

Lab 10: Boot Loading from Flash Memory

Zynq All Programmable SoC

2016.3

Abstract

Booting a system is a critical part of delivering a viable system. Here you will learn how to create and customize a boot image.

Objectives

After completing this lab, you will be able to:

- Create a First Stage Bootloader (FSBL) that works with a custom boot image targeting the QSPI
- Load the image into the QSPI and observe its execution

Introduction

This lab illustrates the steps involved in booting an application from QSPI Flash.

You will be provided with an existing hardware design running on the Zynq® All Programmable SoC PS's Cortex™-A9 processor and a MicroBlaze™ processor. The presence of these two processors will highlight how code from a single Flash device can be configured for multiple processors.

Note: The details of this design can be found in the topic cluster named "Sharing Resources". The important element to remember is that there are multiple processors in this design: a Cortex-A9 and a MicroBlaze processor. Regardless of whether you are using multiple MicroBlaze processors embedded in an FPGA, multiple processors in a PS, or a mix of the two, the basic process remains the same.

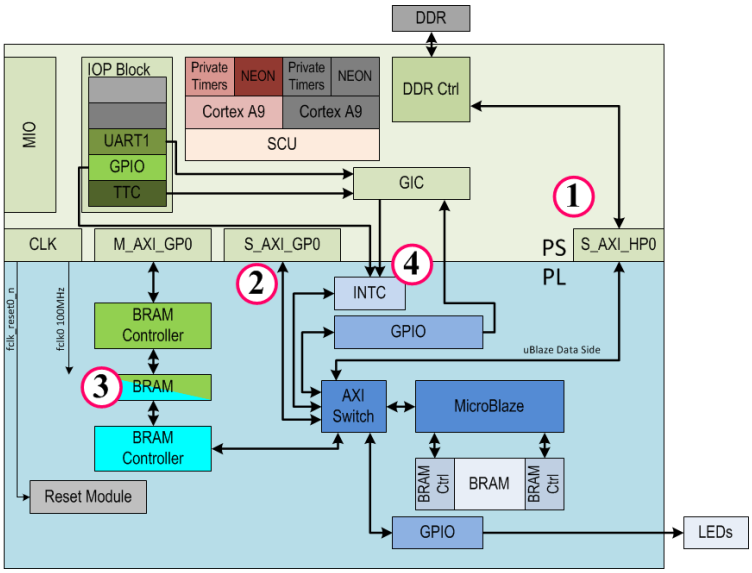


Figure 10-1: Hardware Block Diagram

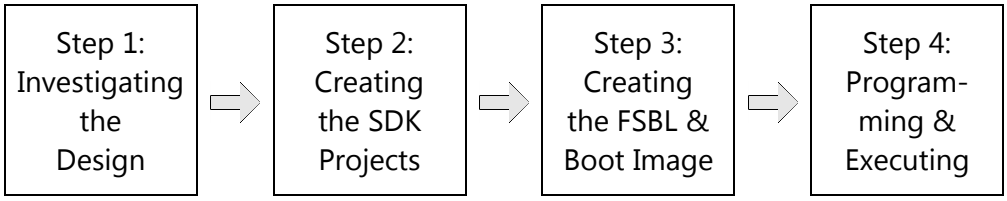
The applications for the processors are typically stored in non-volatile memory (such as QSPI) and is executed from a faster, volatile, off-chip memory resource like DDR or block RAM.

You have been provided with a script to create the hardware project, the pre-generated HDF files as well as the source code and a script to create the SDK projects required to build the boot image.

To simplify development of the boot image, Xilinx SDK provides a Create Boot Image utility. You will use the bootGen tool (available in SDK) to create a bootable image that you will then use to boot the system and verify that the applications have launched.

The applications for this lab are very simple. Both processors initialize themselves then enter a loop where they periodically emit a 'Z' (from the Cortex-A9 processor) or an 'M' (from the MicroBlaze processor). This is only to illustrate that both processors are running their intended code.

General Flow



Investigating the Hardware Design [2016.3] Step 1

A Tcl script has been provided for you to quickly build the hardware project. You will begin by launching the Vivado® Design Suite and running the provided Tcl script to assemble the project.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. Launch the Vivado Design Suite.

This can be done in two standard ways, use your preferred method.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.3** > **Vivado 2016.3**.

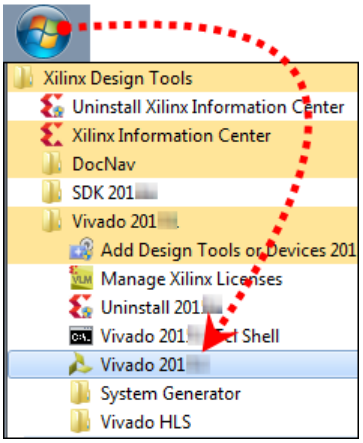


Figure 10-2: Launching the Vivado Design Suite from the Start Menu

-- OR --

Double-click the **Vivado Design Suite** shortcut icon () on the desktop.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

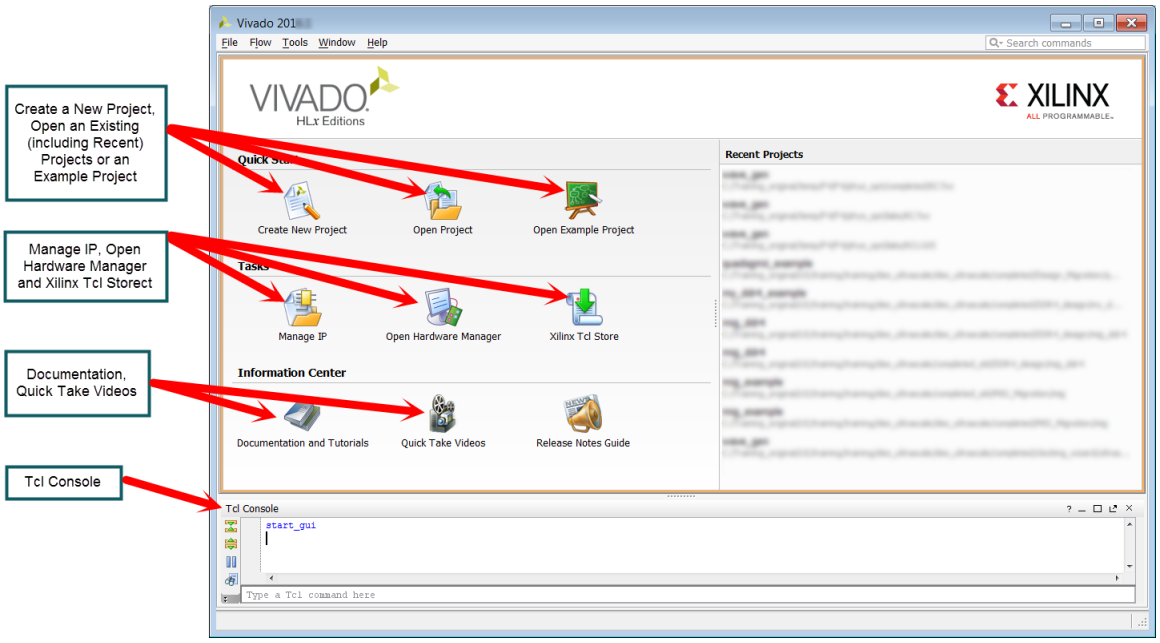


Figure 10-3: Vivado Design Suite Welcome Screen

You will now load the provided Tcl script and have it build the project.

The Vivado Design Suite offers both GUI and scripted control. Scripted control takes the form of Tcl commands. These Tcl commands can be entered directly into the tool one at a time, or an entire Tcl script can be loaded and executed.

1-2. Run a Tcl script.

1-2-1. Locate the Tcl command line entry.

The command line entry can be found either on the Welcome page prior to a project being opened, or once a project has been opened.

From the Welcome screen:

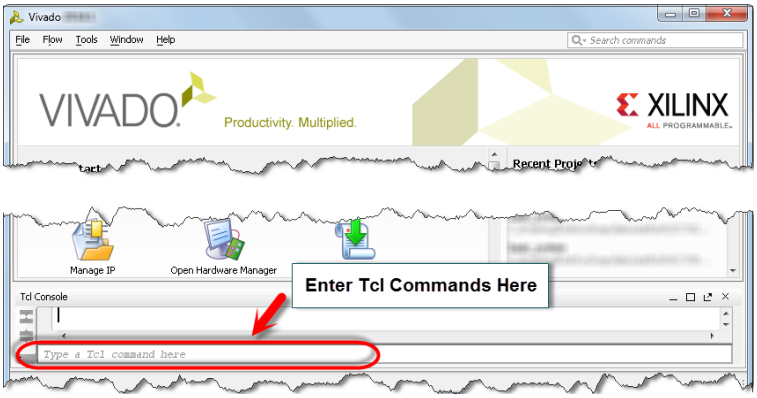


Figure 10-4: Accessing the Tcl Console from the Getting Started Page

From an opened project:

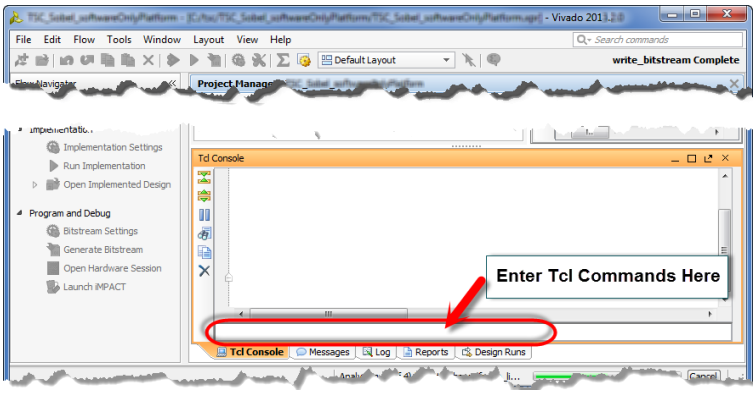


Figure 10-5: Entering Commands into the Tcl Console from an Open Project

The default directory for the Tcl environment is nested within the Xilinx installation directory. This placement, however, is often disadvantageous. In most cases, you will want to navigate to a more useful path. To do this, use the `cd` command to change directory to the user directory.

1-2-2. Change the current working directory to where the Tcl script is located by entering:

```
cd c:/training/bootLoading/support
```

Remember that the Tcl environment is based on Linux and requires the '/' character to delimit hierarchical paths.

1-2-3. Verify that you are now where you want to be by entering the following into the Tcl command line:

```
pwd
```

The current working directory is displayed. If you are not where you want to be, use the `cd` command to change to `c:/training/bootLoading/support`.

1-2-4. Enter the following Tcl command:

```
source bootLoading_completer.tcl
```

The Tcl script is run as though you typed each command included in the Tcl script into the Tcl command line. You can follow the execution of the script and monitor for any errors or warnings in the Tcl Console.

The Tcl script is now loaded and you have been invited to begin entering command and proc names into the Tcl command line.

1-3. Build the project.

1-3-1. Enter the following command to specify the board type, using either ZC702 or Zed as the argument to the `use` proc:

```
use [ZC702 | Zed]
```

You can now specify how the outputs are to be directed.

1-3-2. Specify `base` for using the LEDs on the board:

```
use base
```

1-3-3. Enter the following command to launch the proc that builds the entire project:

```
makeProject
```

The project builds in the Tcl window. When it completes, the Vivado Design Suite GUI will open.

1-4. Investigate the design for the following features:

- **Boot source**
- **Instruction memory for the PS**
- **MicroBlaze processor**
- **Instruction memory for the MicroBlaze processor**
- **Connection between the MicroBlaze processor and the PS**

1-4-1. Click **Open Block Design** in the Flow Navigator window to open the Vivado IP integrator design.

1-4-2. Click the **Show Interface Connections Only** icon to hide clock and reset signals so that you can focus on the AXI connections.

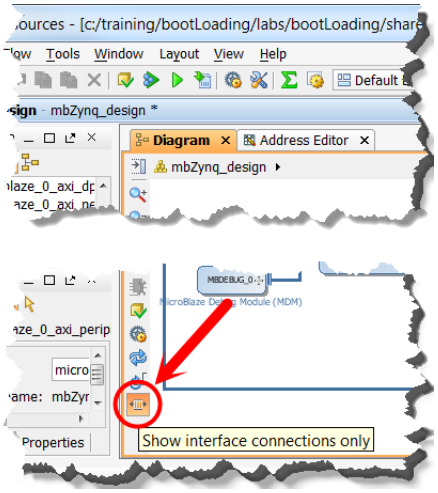


Figure 10-6: Locating the Show Interface Connections Only Icon

Question 1

Where can you find the boot source?

Question 2

What memory is available for the MicroBlaze processor?

Question 3

How are the PS and the MicroBlaze processor connected?

Since no changes should have been made, you can exit the Vivado Design Suite without saving.

1-5. Close the Vivado Design Suite.

1-5-1. Select **File** > **Exit**.

The Exit Vivado dialog box opens.

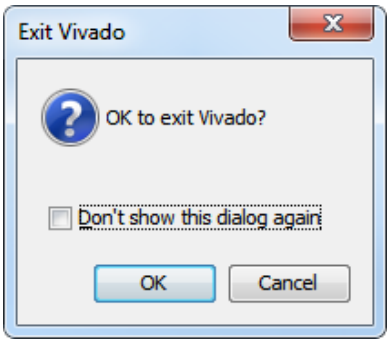


Figure 10-7: Exit Vivado Dialog Box

1-5-2. Click **OK**.

Creating the SDK Projects Step 2

With the hardware investigation now complete, you will move on to briefly investigating the software aspect of the project prior to creating the boot image file.

2-1. Launch the Xilinx Software Command Line Tool (XSCT).

2-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx Software Command Line Tool 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

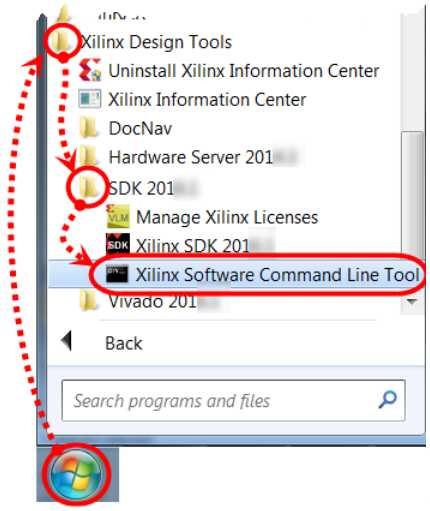


Figure 10-8: Launching XSCT

The Xilinx Software Command Line Tool opens.

Figure 10-9: Xilinx Software Command Line Tool

You can now enter Tcl commands into the tool.

2-2. Build the SDK projects.

2-2-1. From the Xilinx Software Command Line Tool, change directory to the support directory:

```
cd c:/training/bootLoading/support
```

2-2-2. Access the procs needed for the SDK build:

```
source helper.tcl
```

2-2-3. Select the ZC702 or Zed board:

```
use [ ZC702 | Zed ]
```

2-2-4. Create the SDK projects:

```
source SDK_builder.tcl
```

This will create and build the application projects for the Zynq and MicroBlaze processors.

After the script has completed, the Xilinx Software Command Line Tool has no further use and can be closed.

2-3. Launch the SDK tool and set the workspace.

2-3-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

2-3-2. Enter **C:\training\bootLoading\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\bootLoading\lab** directory and click **OK**.

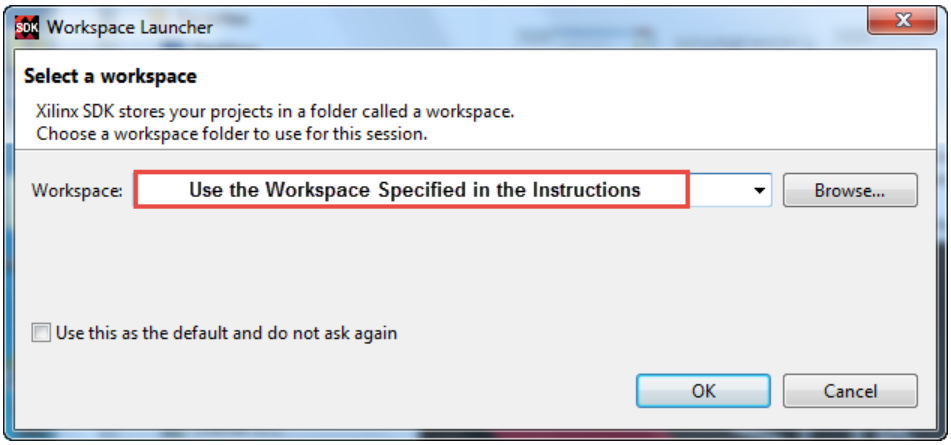


Figure 10-10: Setting Up the Workspace Environment Path

2-3-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

2-3-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

The projects that were just built will appear in the Project Explorer view located in the left side of the SDK window.

Question 4

What projects are in this workspace? What are they for?

Question 5

What information from these projects needs to be included in the boot image?

2-4. Open the BSP settings.

2-4-1. Expand the **sharedResourcesQSPI_Zynq_bsp** project so that the *system.mss* file is shown (1).

The *system.mss* file provides an overview of the BSP and supports modification and regeneration of the BSP and its sources.

2-4-2. Double-click the **system.mss** file to open the Board Support Package Project settings dialog box (2).

2-4-3. Click the **Modify this BSP's Settings** button to open a dialog box that contains the BSP Settings (3).

2-4-4. Click **Overview** in the left window pane (4) if the Overview view is not selected.

2-4-5. Select **xilffs** under the Supported Libraries section (5).

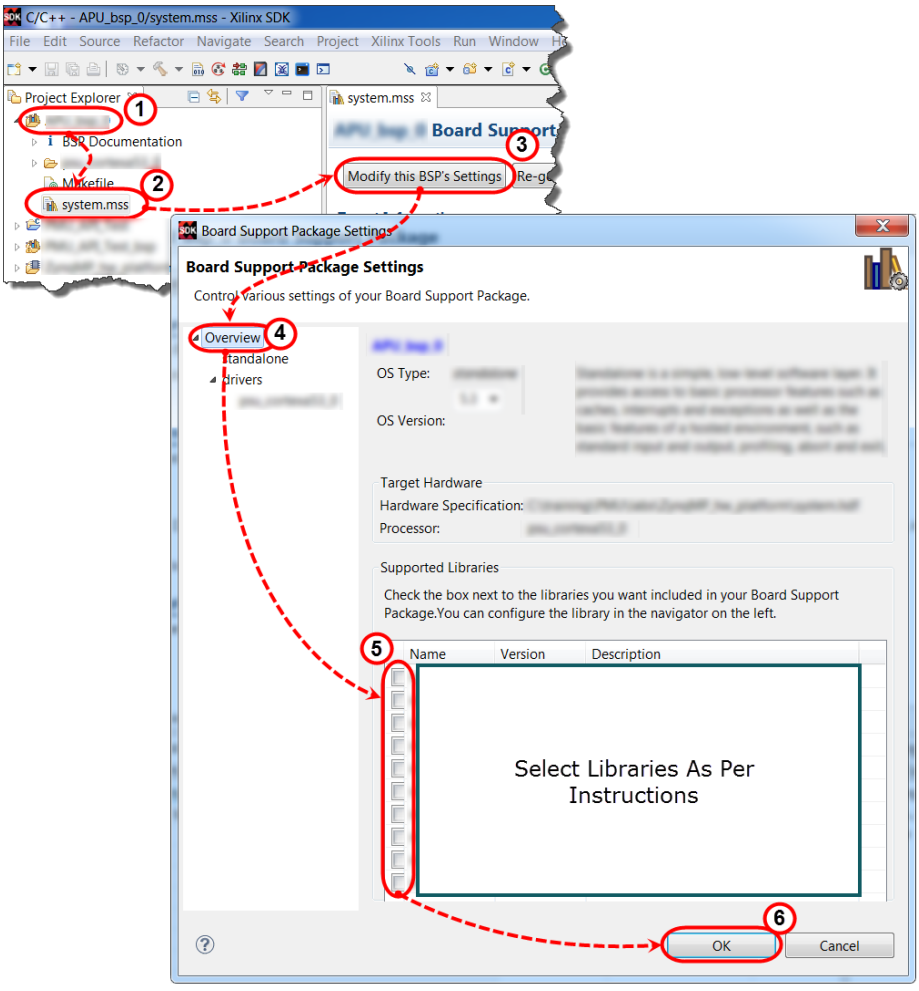


Figure 10-11: Accessing the BSP Customization Dialog Box

2-4-6. Click **OK** to modify the BSP and rebuild the BSP's sources and binaries (6).

Note: The xilffs library is needed by the first stage bootloader application.

Creating a Boot Image File Step 3

You will now create the First Stage Bootloader application that will execute on the Cortex A9_0 processor and take over booting from the bootrom program.

Using the Application Project Wizard is a quick way to set up a C or C++ software application project that targets an existing processor and OS platform (Standalone or Linux). You can automatically generate the board support package (BSP) or select an existing one. Based on the dialog box choices, the appropriate tool chain is selected for pre-processing, compiling, assembling, and linking.

3-1. Create a new C/C++ application project named *Zynq_FSBL_app*. Use the board support package named *Zynq_FSBL_bsp*.

3-1-1. Select **File** (1) > **New** (2) > **Application Project** (3) to open the New Project dialog box.

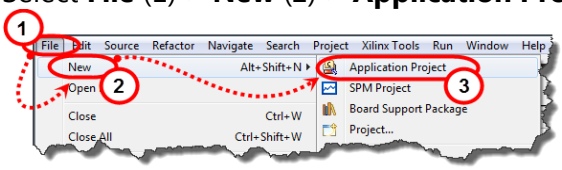


Figure 10-12: Creating an Application Project

3-1-2. Enter **Zynq_FSBL_app** as the project name.

3-1-3. Ensure that you have **sharedResourcesQSPI_hw** selected from the Hardware Platform drop-down list as the SDK tool can manage multiple platforms within a single workspace.

This will populate the Processor drop-down list accordingly.

3-1-4. Ensure that **ps7_cortexa9_0** is selected from the Processor drop-down list.

3-1-5. Ensure that **standalone** is selected from the OS Platform drop-down list.

3-1-6. Select **Use Existing** and choose an existing BSP from the drop-down list if you have already created a BSP for this hardware platform; otherwise, select **Create New** and enter the name for the BSP as **Zynq_FSBL_bsp**.

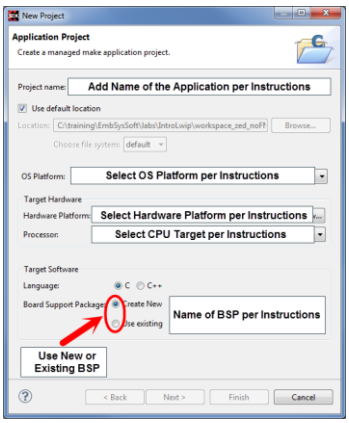


Figure 10-13: Entering Application Project Information

3-1-7. Click **Next** to select the template for this application.

3-1-8. Select **Zynq FSBL** (1).

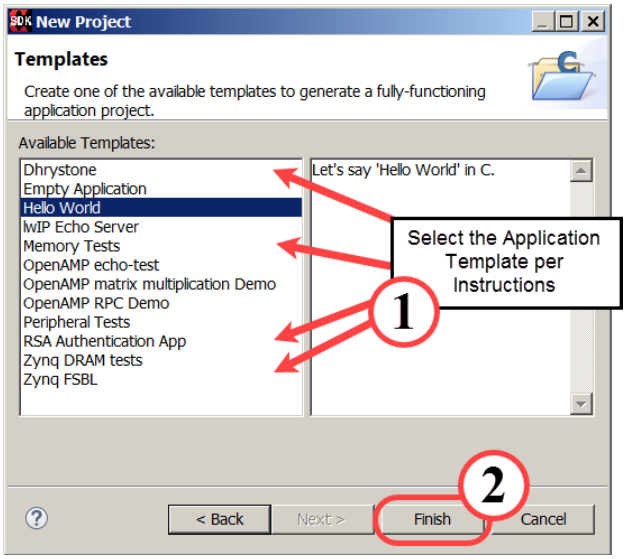


Figure 10-14: Selecting an Application Template (Selection in Figure May Not Match Instructions)

3-1-9. Click **Finish** to create the new project (2).

As the project is created, the new BSP is compiled and any sources from the templates are also compiled. The creation of the new project usually takes less than a minute.

3-2. Build all of the projects.

3-2-1. Force the building of all applications by selecting **Project** > **Build All**.

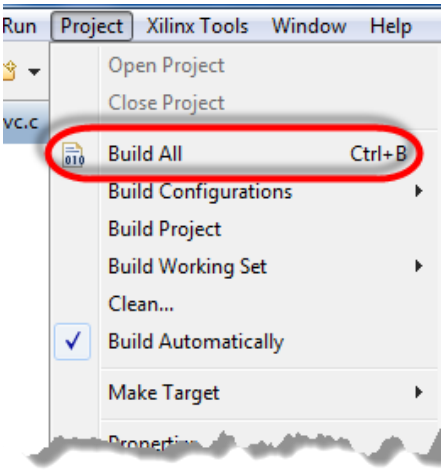


Figure 10-15: Selecting Build All

Note: In 2016.3 the SDK tool does not automatically build the FSBL project upon creation; hence, the need to build all projects.

After the FSBL application has been created, you will create a boot image file that contains the Zynq All Programmable SoC FSBL, the BIT file for programmable logic (PL) configuration, and the software application ELF file.

3-3. Create the boot image.

3-3-1. Right-click the **Zynq_FSBL_app** project from the Project Explorer view to open the context menu.

3-3-2. Select **Create Boot Image** to open the Create Boot Image Wizard.

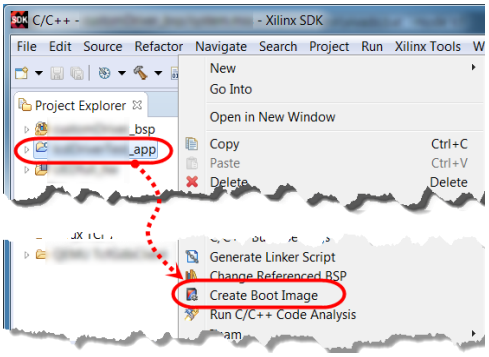


Figure 10-16: Selecting Create Boot Image

The Create Zynq Boot Image dialog box opens.

3-3-3. Make sure that **Create a new BIF file** is selected (1).

The *Output BIF file path* and the *Output path* field will auto-populate (2, 3).

The Boot image partitions section will auto-populate based on the bitstream provided in the hardware specification project and the ELF from the *Zynq_FSBL_app* application project.

Note: If there is a bitstream file in the design, it must be the next entry after the bootloader and any other files should be after the bitstream file.

You will now have to modify the bitstream image and add the remaining file to the boot image partitions.

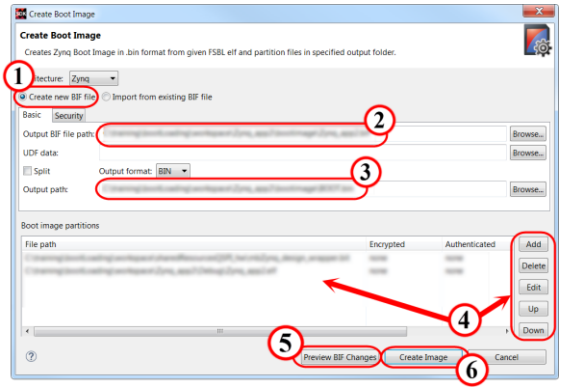


Figure 10-17: Creating the Boot Image File Using the Create Boot Image Wizard

3-3-4. Select the second entry in the *Boot image partitions* box *C:\training\bootLoading\lab\sharedResourcesQSPI_hw\mbZynq_design_wrapper.bit* (4).

3-3-5. Click the **Edit** button (4) to edit the bitstream entry.

3-3-6. Using the **Browse** button next to the *File path* field, browse to *C:\training\bootLoading\lab\sharedResourcesQSPI_hw*.

3-3-7. Double-click **mbZynq_design_wrapper_download.bit** to select and add the file.

3-3-8. Click **OK** to add this modified partition to the boot image.

Note: This bitstream has been modified to preload the MicroBlaze processor with the MicroBlaze processor application. If this were not done, the MicroBlaze processor block RAM would be loaded with all zeros and do nothing. As this is a simple design, the functionality to load the MicroBlaze processor code from the PS is not available. Details of how this bitstream is generated can be seen in the last line of the *SDK_builder.tcl* script.

3-3-9. Click **Add** to begin adding the next ELF file (4).

This will open an *Add new boot image partition* window.

3-3-10. Using the **Browse** button next to the *File path* field, browse to *C:\training\bootLoading\lab\Zynq_app\Debug*.

3-3-11. Double-click **Zynq_app.elf** to select and add the file.

While there are other options including encryption, alignment, offset, etc., you can leave these blank for this lab.

3-3-12. Click **OK** to add this partition to the boot image.

3-3-13. [Optional] Click **Preview BIF Changes** (5).

If you had a previous BIF file, this would compare the old BIF file (left-hand pane) with the newly generated BIF file (right-hand pane).

Question 6

How does the boot image file indicate what gets loaded first and where?

Question 7

How might you add a data file to the boot image?

3-3-14. If you examined the BIF file, click **OK** to close the comparison window.

3-3-15. Click **Create Image** to build the boot image (6).

3-4. Review the generated file.

3-4-1. Expand the **Zynq_FSBL_app** folder using the Project Explorer tab.

Note that the *bootimage* folder has been created.

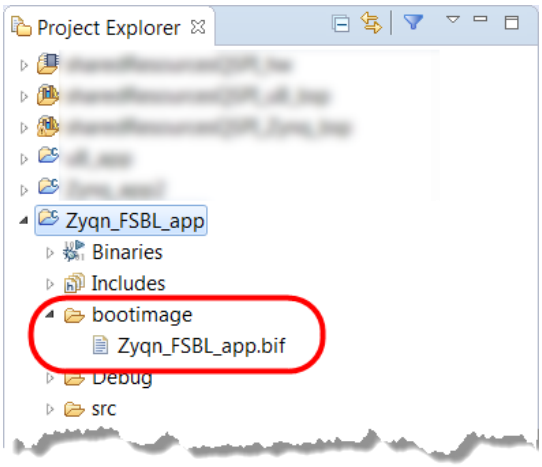


Figure 10-18: bootimage Directory

The *BOOT.bin* image is used for booting the application from Flash and will appear under *bootimage*.

3-4-2. Double-click **Zynq_FSBL_app.bif** and review the contents.

The image shows the partitions and order in which the files will be executed. This is the same information you saw when you clicked the **Preview BIF Changes** button.

Note: The boot image file can also be created manually, the details of which are outside the scope of this lab.

Programming the Flash and Executing the Application from Flash Step 4

The boot image file created in the previous step will be used to program the Flash. You will then power-cycle the board and boot the system from the Flash device.

For this step, ensure that the board is properly connected to the computer, that there is no SD Card inserted, and that the development board is powered on.

4-1. Select the boot image file to program the Flash memory.

4-1-1. Select **Xilinx Tools** > **Program Flash**.

4-1-2. Click **Browse** to locate the boot image.

4-1-3. Browse to the *C:\training\bootLoading\lab\Zynq_FSBL_app\bootimage* directory.

4-1-4. Select the **BOOT.bin** image file.

4-1-5. Click **Open** to use this file as the source.

For this lab, there is no offset that is required.

4-1-6. Select **qspi_single** from the Flash Type drop-down list.

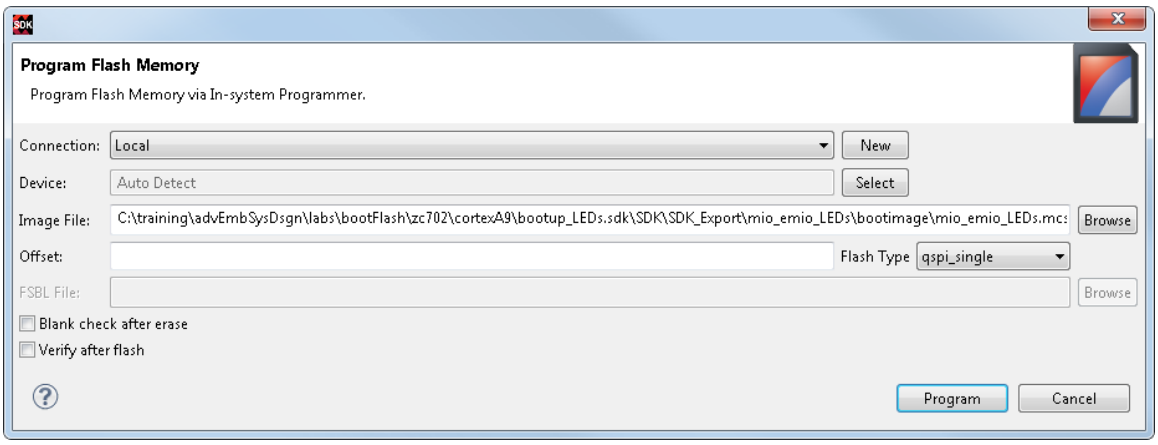


Figure 10-19: Programming the Flash

4-1-7. Click **Program**.

This process will take a few minutes to complete.

You can see the programming status in the Console window.

You will see the following message once the programming is completed:

Flash Operation Successful

4-2. Determine the COM port number for the USB serial port used on the board and configure the related terminal emulator program for that COM port at 115200-N-8-1.

If you do not recall how to perform this task, refer to the "Configuring the SDK Terminal" section under SDK Operations in the *Lab Reference Guide*.

4-3. Set and verify the jumper settings for booting from flash.

4-3-1. Power off the board now that you have programmed the flash, as you need to change the jumper settings for the flash boot.

4-3-2. **ZC702 board users:** Make sure that the jumper settings for booting from flash are set as shown below.

If the jumpers have not been populated, set the switch SW16 to 00010. The jumpers and switch are connected to the same nets.

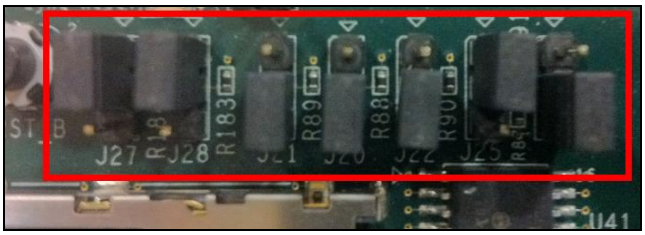


Figure 10-20: Jumper Settings – Flash Boot Mode

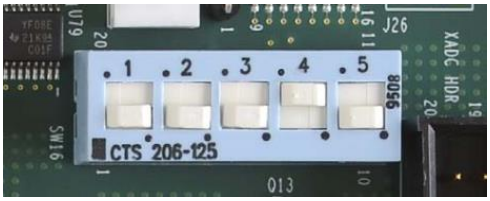


Figure 10-21: SW16 Configured to Boot from QSPI

ZedBoard users: Make sure that the jumper settings for booting from flash are set as shown below.

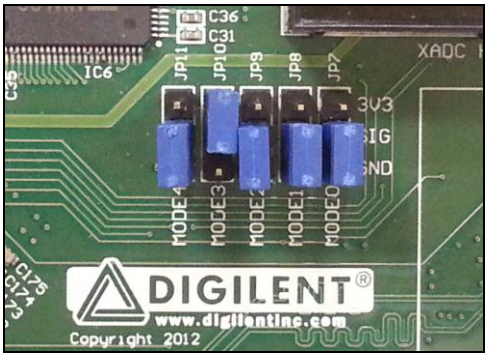


Figure 10-22: Jumper Setting for QSPI Boot (ZedBoard)

4-4. Connect the ZC702 or ZedBoard to your machine.

4-4-1. Make sure that both the UART/USB and programming USB cables are connected to your machine.

4-4-2. Power the board off now if it is powered on.

4-5. Verify the running software application program.

4-5-1. Power on the board.

You should see the DONE LED go ON. The application will now be loaded.

4-5-2. Verify the messages on the serial port console.

You should see a mix of 'Z' (indicating that *Zynq_app* is running) and 'M' (indicating that *uB_app* is running). These may take a up to 10 seconds to appear.

Question 8

What is the ratio of 'Z's to 'M's?

_____

_____

_____

4-6. Exit the SDK tool and power off the board.

4-6-1. Select **File** > **Exit**.

4-6-2. Power off the development board.

Summary

You just reviewed a hardware and software embedded project and created a bootable image that you burned into the target board's Flash memory. You learned how to create various boot image sections and identify the sections as ELF, BIT, data, and bootloader.

This process can be applied to significantly more complex systems, including those with Second Stage Boot Loaders and operating systems.

Answers

1. Where can you find the boot source?

   Since this is a Zynq device, booting is controlled from a series of jumpers that is read when the device is powered up. The board should be configured to boot from the QSPI device. This means that the pins in the MIO should be connected to the QPSI device and should be enabled.

   You can quickly check to see how the MIO is configured by double-clicking the PS block and selecting either Peripheral I/O Pins or MIO Configuration.

2. What memory is available for the MicroBlaze processor?

   The MicroBlaze processor is connected to the HP port of the PS, giving it access to DDR memory, as well as being equipped with its own local block RAM memory for cache and low-latency access.

3. How are the PS and the MicroBlaze processor connected?

   There are two points of connection: an AXI path to the HP port of the PS for access to the DDR and a master AXI connection into the PS so that the MicroBlaze processor can access the IOP block within the PS. This is how the MicroBlaze processor will issue serial characters.

4. What projects are in this workspace? What are they for?

   - *sharedResourcesQSPI_hw:* Hardware project that describes the environment in which the software will operate. This includes the number of types of processors, types and quantities of memory, types and quantities of peripherals, and a memory map placing all devices within the memory scheme.
   - *sharedResourcesQSPI_uB_bsp:* Board support package derived from the hardware project specific to the MicroBlaze processor. This includes device drivers for the peripherals and processors, utilities, and other important support features.
   - *sharedresourcesQSPI_Zynq_bsp:* Board support package derived from the hardware project specific to the Cortex-A9 processor in the PS. As with the MicroBlaze processor's BSP, this includes device drivers, utilities, and other necessary support.
   - *uB_app:* Application that runs on the MicroBlaze processor.
   - *Zynq_app:* Application that runs on the Cortex-A9 processor in the PS.
   - *Zynq_FSBL_app:* First Stage Boot Loader that will run on the Cortex-A9 processor in the PS. The job of this application is to load the ELF files into their appropriate places in memory and load the bitstream in the PL.

5. What information from these projects needs to be included in the boot image?

   Typically the bitstream for the PL is included in the hardware project and will be needed for the boot image. When the applications are compiled, the necessary information is extracted from the BSP, so the BSP projects are not explicitly used. The BSP combined with the application for the Zynq device and MicroBlaze processors produces a pair of ELF files: one for the Cortex-A9 processor and the other for the MicroBlaze processor.

   A mapping of how the two ELF files and bitstream is also required and will be created in the next step as well as a First Stage Boot Loader to extract the various types of files from the NV-RAM device and place them in memory.

6. How does the boot image file indicate what gets loaded first and where?

   The BIF file tags the bootloader with a [bootloader] stamp. This file must be the first ELF file in the Boot Image Partitions list. This was automatically determined by the tools. The partition type could also be specified in the Edit Partition dialog window. The FSBL contains information regarding how to handle BIT files and ELF files.

7. How might you add a data file to the boot image?

   Click **Add** to create a new boot image partition, browse to your data file, and select datafile from the Partition type drop-down list.

8. What is the ratio of 'Z's to 'M's?

   There should be two 'M's appearing for every one 'Z' because *uB_app* emits an 'M' every five seconds and *Zynq_app* emits a 'Z' every 10 seconds.

Lab 11: SDK Tool Profiling

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

This lab demonstrates how to profile a program, interpret reports, and verify performance with multiple calls. You will use profiling information to determine which tasks are consuming the most amount of time. The instructions will guide you through profiling a program and analyzing the results.

Objectives

After completing this lab, you will be able to:

- Profile an application by using the SDK tool
- Profile code and determine the most time-consuming functions via the SDK tool
- Thoroughly interpret the profiling results

Introduction

Profiling is a method by which the software execution time of each routine is determined. Routines that are frequently called are best suited for placement in fast memories; cache being one candidate.

TCF Profiler is a non-intrusive profiler based on PC sampling. An application is run that collects the list of PC recordings and maps them to the locations of subroutines. The number of occurrences of all of these subroutines is summed and reported to the user.

The following is an example of a non-intrusive profiler.

Given a main and a number of functions, TCF Profiler samples the functions using the PC.

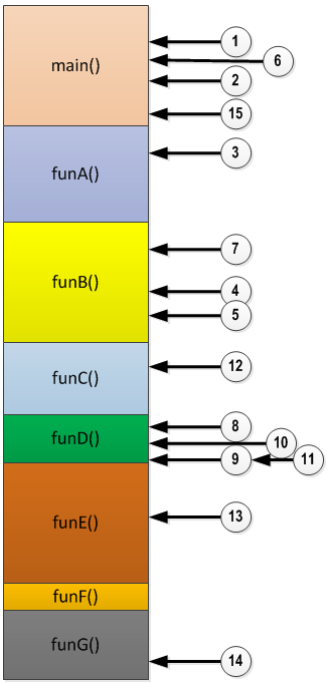


Figure 11-1: Example of Profiling Interrupts Against a Memory Map

After a sufficient amount of time passes and a statistically significant number of samples is collected (15 samples stated in this example is NOT enough; this is just an example), the data is extracted from the system memory.

function	freq
main()	4
funA()	1
funB()	3
funC()	1
funD()	4
funE()	1
funF()	0
funG()	1

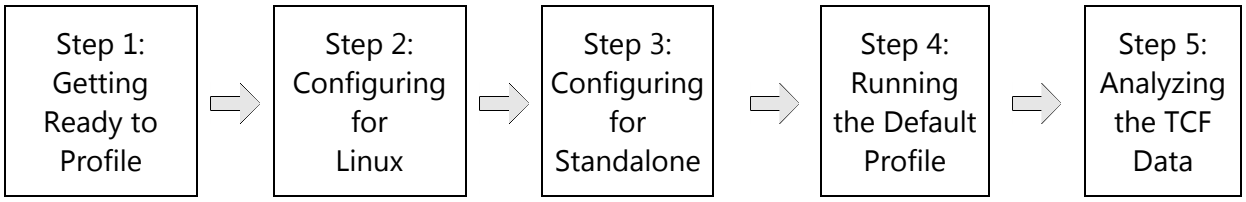
Figure 11-2: Frequency Table for Above Example

This information can be represented graphically, in terms of time, etc. One should note a few things from this example. First, funF() was never called. Really? Or were too few samples captured? Perhaps the sample interval was set too long and funF() executes very quickly and was simply never captured there?

The profiling tools enable you to set the granularity of the timer; but be careful, too fine a granularity might impact your code in other ways. If this is rerun, it is entirely possible that a slightly different set of results may be generated. This becomes a probabilistic analysis.

Neither profiling technique is perfect, but they do not need to be. Remember that profiling will give you a *feel* for how your code is executing, not a perfect representation.

General Flow



Getting Ready to Profile — Step 1

You will begin by creating an SDK tool software application. The provided C source code will be used as the target for profiling. The software application will compile and link without errors.
You will use *C:\training\SDK_Profiling\lab* as the SDK tool workspace. The SDK development environment will automatically create a hardware platform depending on the development board that is selected.
Once the SDK tool is launched and a workspace is set up in this directory, you will not be able to move these directories.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\SDK_Profiling\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\SDK_Profiling\lab** directory and click **OK**.

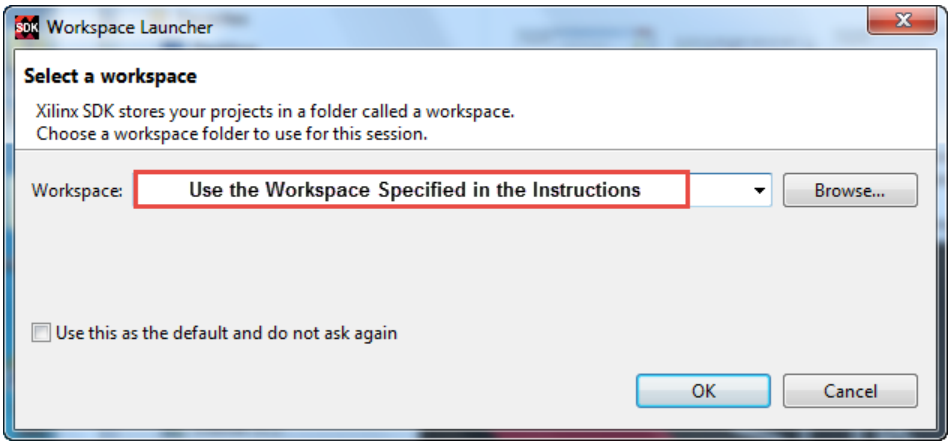


Figure 11-3: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

If you are working in the SDK tool and do not have a hardware project and BSP specified, select the BSP and hardware project during the import process. SDSoC tool projects do not require a hardware platform nor a BSP.

One or more projects can be exported as a single zipped file. This is convenient when passing projects or parts of projects to teammates or clients. Once the recipient has received this archive, the zip file must be processed and one or more projects can then be imported.

1-2. Import an existing project.

1-2-1. Select **File** > **Import** to open the Import Wizard.

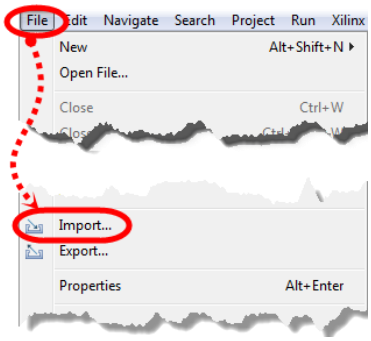


Figure 11-4: Accessing the Import Wizard

The Import dialog box opens.

1-2-2. Expand the **General** node to access the commonly used methods (1).

1-2-3. Select **Existing Projects into Workspace** as the goal is to import an existing project (2).

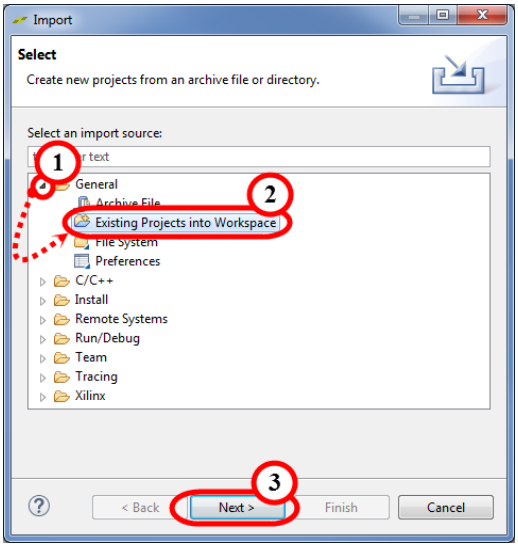


Figure 11-5: Choosing to Import an Existing Project into the Workspace

1-2-4. Click **Next** to enter project-specific data (3).

1-2-5. Select the **Select archive file** option (1).

Note that projects can be archived either as single zip files, or you can import from a directory. Typically, projects are preserved as archives as they are easier to move.

1-2-6. Click **Browse** to navigate to *C:\training\SDK_Profiling\support* (2).

1-2-7. Select **profLab_{standalone | linux)_(ZC702 | Zed)_start.zip**, which contains the archived projects.

1-2-8. Click **Open** to open the archive and list the various projects in that archive.

1-2-9. Select **all of the listed projects** to import (3).

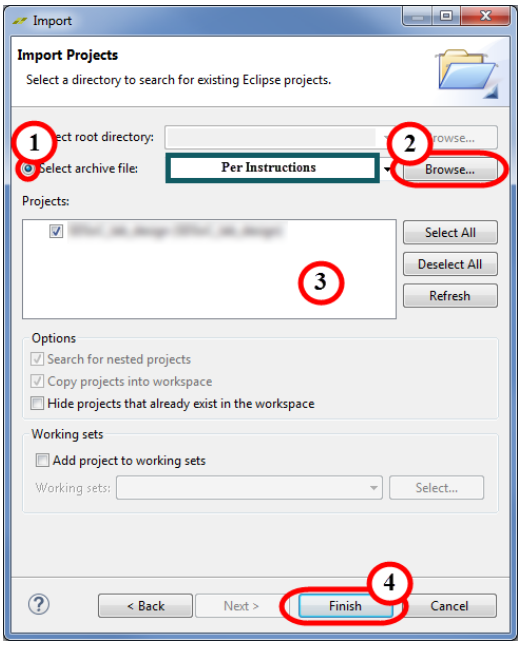


Figure 11-6: Import Settings for Archived Projects

1-2-10. Click **Finish** to perform the importing of the project (4).

Because the profiling operation slows program execution speed, the source code will be modified to ensure that the profiling completes in an acceptable time frame.

1-3. Open *SDSoC_lab_design_main.c* in the editor.

1-3-1. Locate **SDSoC_lab_design_main.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

1-3-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

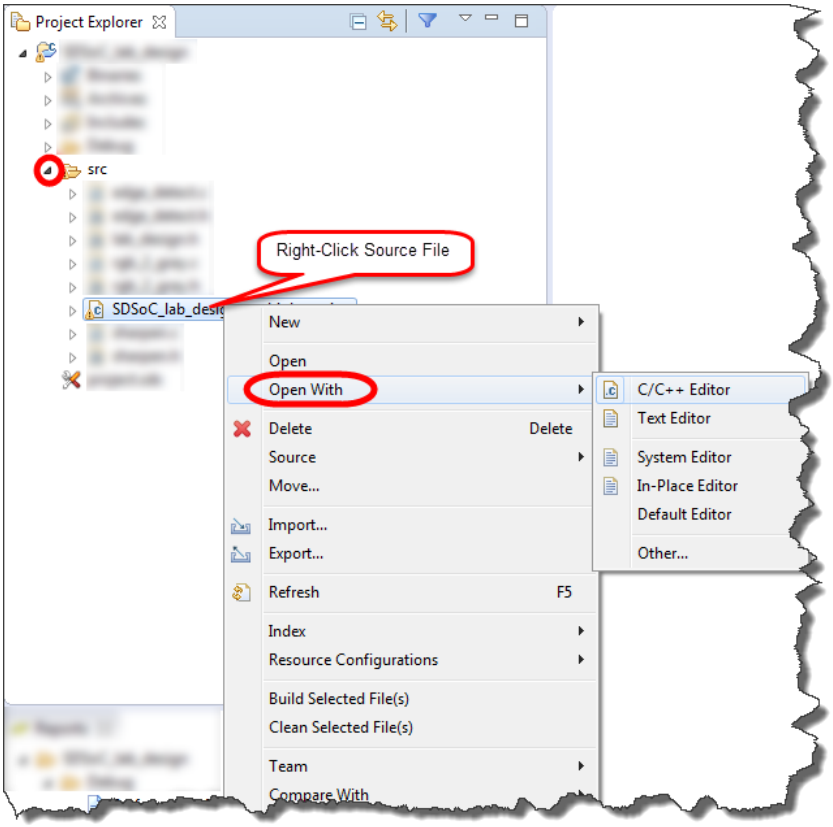


Figure 11-7: Opening a Source File via Right-Click

1-4. Find *#define LOOPS*.

The find/replace operation is accessible through both a click sequence and a keyboard shortcut.

1-4-1. Select **Edit** > **Find/Replace** or press <**Ctrl** + **F**>.

The Find/Replace dialog box opens.

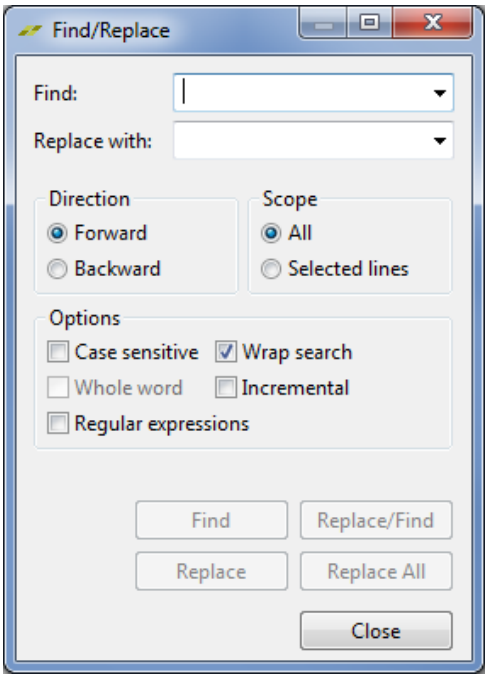


Figure 11-8: Default Find/Replace Dialog Box

1-4-2. Enter **#define LOOPS** in the Find field.

1-4-3. Click **Find** or press <**Enter**> to find the next occurrence of *#define LOOPS*.

1-4-4. Continue clicking **Find** or pressing <**Enter**> until you locate the specific instance that you are looking for.

With the **Wrap search** option enabled, the file is treated as a continuous loop and the find operation will jump to the next occurrence at the top of the file (when searching forwards) or at the bottom of the file (when searching backwards). A "ding" sound is made when the search wraps around.

If you are looking for text within a specific region of the code you would first highlight the region to perform the search in, then launch the Find/Replace function as described in this topic.

1-4-5. Click **Close** to close the Find/Replace dialog box.

1-5. Set the macro *LOOPS* to 1.

This sets how many times to perform the image processing operations on the image.

For profiling in Standalone with stack tracing enabled (which this lab does), the program execution will be ~12 times slower than running normally. This is because the JTAG access to certain registers to gather performance data is limited by the JTAG cable speed. The profiling is still non-intrusive but the execution time will be reduced.

1-5-1. Edit this line to read:

#define LOOPS 1

Tip: Do not insert ';' at the end of this macro as this will make it difficult to diagnose compilation errors.

1-5-2. Save the file.

1-6. Find *return 0*.

You are finding return 0 to add a breakpoint here. With the Linux versions of the lab, when the profiling is complete, the debug session will end and the newly acquired profile data will be discarded. The breakpoint is to keep the debug session *active* so that the profile information can be inspected.

1-6-1. Using the technique you used previously, find the text "**return 0**".

Note: This is the final statement in the *main()* function.

1-7. Add a breakpoint at the current line.

1-7-1. Right-click the left margin in the editor at the line at which to insert a breakpoint.

1-7-2. Select **Add Breakpoint** or **Toggle Breakpoint**.

If you select Add Breakpoint, then the Breakpoint Properties dialog box opens. Here you can specify the details for this breakpoint if necessary to create a conditional breakpoint.

If you select Toggle Breakpoint, then a simple breakpoint will be created (or removed if one already existed) on the line you specified using default breakpoint properties.

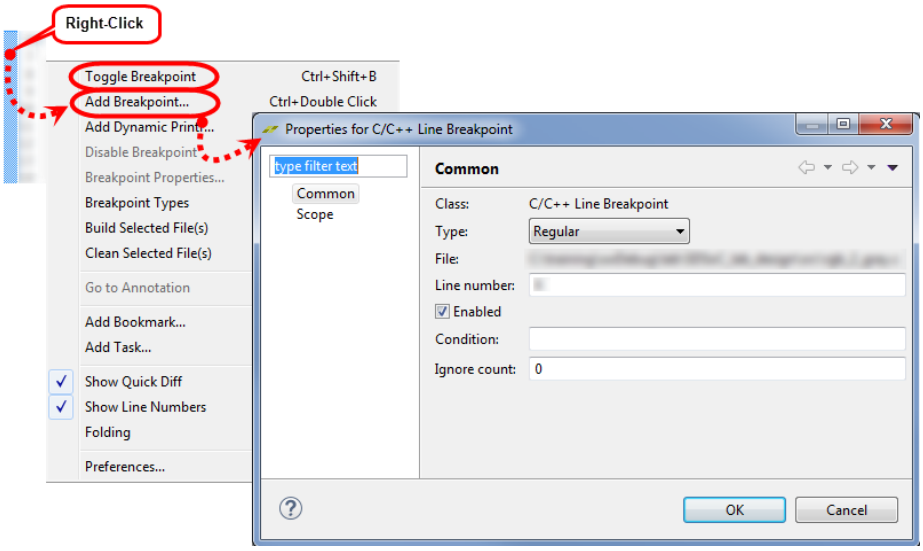


Figure 11-9: Adding a Simple or Conditional Breakpoint

1-7-3. Click **OK** to create the breakpoint if Add Breakpoint is selected.

Note: This breakpoint is added to ensure that the profiling results are not automatically cleared when the program completes execution.

A Debug configuration defines how you want the system to work when performing a debug operation. Typically a debug operation switches to the Debug perspective. While there are a significant number of switches and options, the most common are shown below.

1-8. Set the build configuration to Debug.

1-8-1. Right-click the project name in the Project Explorer pane to open the context menu (1).

1-8-2. Select **Build Configurations** to view the options for building the different types of configurations (2).

1-8-3. Select **Set Active** to see the available configurations (3).

1-8-4. Select **Debug** to build the desired configuration (4).

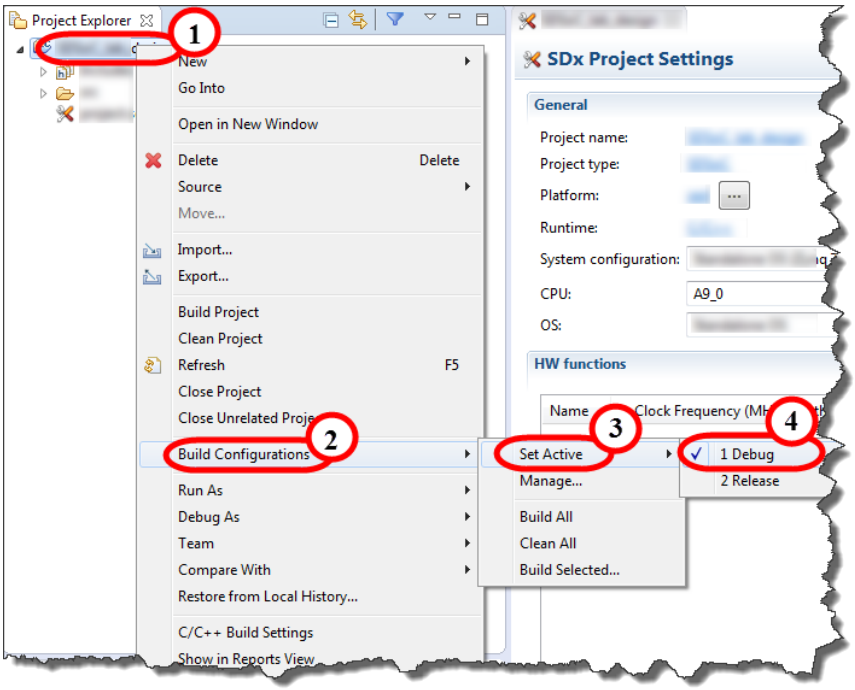


Figure 11-10: Selecting the Debug Build Configuration (Example)

1-8-5. Right-click the project name and select **Build Project**.

This will take a minute or two to complete building the project.

Note: This is to recompile the newly modified source code. It will be recompiled with the SDDebug configuration for the SDSoC tool or Debug for the SDK tool and should take approximately two minutes.

Standalone users: You can skip to the "Configuring the SDK Tool for Standalone" step.

Linux users: In the SDSoC tool, once any configuration has been run (with the default *Build SD Card image* option on), then the tools build an SD card image that is located under *<project name> > [SDDebug | SDRelease]*.

Linux requires the board to already have been booted with Linux; otherwise the debug session will fail. There are three solutions to this issue. First, the project can be built as above and the SD card can be prepared. Second, you can allow the debug session to fail and then prepare the SD card, reboot the board, and relaunch the debug session. Third, you can have a generic Linux image already running on the board.

As the project has just been compiled and there is a generated Linux image ready to be copied to an SD card, this will be used to configure Linux on the development board so that the debug session will not fail on the first try.

For the SDK Linux project, there is no automatically generated Linux SD card image. This has been provided as part of the lab material.

Configuring the SDK Tool for Linux Step 2

Several tasks must be performed in order to successfully run Linux. These tasks may have been performed for you by your training provider in a classroom environment. If you are outside of this environment, you can refer to the following topics in the *Lab Setup Guide*:

- Configure the jumper settings on the board to boot from the SD card.
- Set the host's Ethernet address to a static IP (suggested: 192.168.1.11).
- Identify the proper COM port (must be done with the board powered on).

Note: If you have a preconfigured SD card available as part of the lab, the following *Preparing an SD card* instruction is optional. If you do not have a preconfigured SD card, it may be necessary to build the project and generate an SD card image (if using the SDSoC development environment) or copy from the default directory as shown below.

Linux must boot from a properly configured SD card. Many tools generate the necessary file images for Linux, FreeRTOS, and standalone that can be simply copied from a specified location on the host machine to the SD card.

2-1. Prepare an SD card.

2-1-1. Insert an SD card into the PC's SD card slot.

2-1-2. Browse to the SD card drive using Windows Explorer.

Optional: You may want to erase or reformat the SD card at this time, which may prevent any unwanted interaction with other files that may be on the card.

2-1-3. Open a second Windows Explorer window to browse to the files that you will copy to the SD card.

2-1-4. Browse to the image located at *C:\training\SDK_Profiling\support\Standard_[ZC702 | Zedboard]_SDCard_Linux_Image.zip*.

2-1-5. Drag-and-drop all the files from the source directory to the SD card.

Note: The files should go into the root of the SD card.

2-1-6. Close both Windows Explorer windows.

2-1-7. Remove the SD card from the PC card slot.

2-1-8. Turn off power to the hardware platform.

Note: The boot selection switches or jumpers must be properly set to boot from the SD card. See the appropriate section in the *Lab Setup Guide*.

Insert the SD card into its slot on the hardware platform.

2-2. Apply power to (turn on) the ZC702 or ZedBoard hardware platform.

2-2-1. Make sure that AC power is connected to the power brick.

2-2-2. Slide the power switch to the on position.

Some LEDs on the board will illuminate when the board is powered.

2-3. Launch and configure your serial port terminal emulator.

The Terminal tab can be used for the ZC702 board; however, there is an issue with drivers when information is sent to a ZedBoard. Tera Term is a freeware serial port terminal emulator that has been tested successfully with the target boards.

2-3-1. Identify the COM port associated with your board.

If you are unfamiliar with this process, refer to the *Lab Setup Guide*.

2-3-2. Set the serial port connection within the serial port terminal emulator to **115200** baud, **8** data bits, No parity, **1** stop bit.

If you are unfamiliar with how to use Tera Term, refer to the *Lab Setup Guide*.

If you are unfamiliar with how to use the Terminal tab, refer to the *Lab Reference Guide* under either **SDSoC** or **SDK Tool Operations** > **Configuring the Terminal**.

Now that the board is powered on and Linux has finished its boot process, the Linux environment must be configured by using the serial port emulator terminal.

Note: You may have to wait for a few minutes for Linux to boot.

2-4. Log in to Linux.

2-4-1. Log in with the user name and password of **root** / **root** when prompted in the serial port terminal.

2-5. Set the IP address of the board.

2-5-1. Enter the following at the Linux command prompt to change the IP address of the board to 192.168.1.10:

```
ifconfig eth0 192.168.1.10
```

Note: This can be any address other than the one that the host is configured to.

Optional: You can verify the IP address by entering the following command:

```
ifconfig eth0
```

The response should be *similar* to the following:

```
eth0    Link encap:Ethernet  HWaddr 00:0A:35:00:01:22
        inet addr:192.168.1.10  Bcast:192.168.1.255
Mask:255.255.255.0
        inet6 addr: fe80::20a:35ff:fe00:122/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:23 errors:0 dropped:0 overruns:0 frame:0
        TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:2913 (2.8 KiB)  TX bytes:1446 (1.4 KiB)
        Interrupt:54 Base address:0xb000
```

2-6. Verify the Ethernet connectivity between the host and the development board.

This will also indirectly verify that the host PC Ethernet port is set to an IP address of 192.168.1.11.

2-6-1. Ping the host using the Linux terminal console to verify connectivity between the host and the target:

```
ping 192.168.1.11 -c 1
```

If the ping was successful (indicated by a 0% packet loss with roundtrip times), you can continue to the next section.

If it was not successful, there are several possibilities:

- The host IP address is not set properly.
  - See instructions for configuring the host's IP address in the *Lab Setup Guide*.
- The Ethernet cable may not be properly connected.
  - Unplug and replug the cable on both ends; verify that the Ethernet LEDs are flickering (if available).
- The Windows firewall is blocking the connection. Follow the procedure below to disable the Windows firewall.
  - Access the Windows Firewall controls through the Control Panel (select **Start** > **Control Panel** > **System and Security** > **Windows Firewall**).
  - From the options in the left sidebar, select **Turn Windows Firewall on or off**.
  - Select the **Turn off Windows Firewall** option for both network location settings.
- The sub-net address for either the board or host may not be entered correctly.
  - If you need to configure your PC's Ethernet port, refer to "Configuring the PC's Ethernet Port for Remote System Explorer" section under SDK Operations > Linux and Remote System Explorer in the *Lab Reference Guide*.

Configuring the SDK Tool for Standalone Step 3

A couple of tasks must be performed in order to successfully run Standalone. These tasks may have been performed for you by your training provider in a classroom environment. If you are outside of this environment, refer to the following topics in the *Lab Setup Guide*:

- Configure the jumper settings on the board to boot from JTAG.
- Connect and power up the board.
- Identify the proper COM port (must be done with the board powered on).

3-1. Apply power to (turn on) the ZC702 or ZedBoard hardware platform.

3-1-1. Make sure that AC power is connected to the power brick.

3-1-2. Slide the power switch to the on position.

Some LEDs on the board will illuminate when the board is powered.

3-1-3. Connect the JTAG and UART cable from the host PC to the laptop.

3-2. Launch and configure Tera Term (or equivalent serial port terminal emulator). The SDx tool Terminal tab can be used for the ZC702 board; however, there is an issue with drivers when information is sent to a ZedBoard.

3-2-1. Identify the COM port.

If you are unfamiliar with this process, refer to the *Lab Setup Guide*.

3-2-2. Set the serial port connection to **115200 8N1**.

If you are unfamiliar with how to use Tera Term, refer to the *Lab Setup Guide*.

Now that the board is powered on, it is ready to be programmed.

Running the Default Profile Step 4

For Standalone users, proceed with the instruction below. For Linux users, skip to instructions below that begin with "For Linux users, proceed with the instructions below."

A Debug configuration defines how you want the system to work when performing a debug operation and maps an ELF object file to a target for execution. Typically, a debug operation switches to the Debug perspective. While there are a significant number of switches and options, the most common are shown below.

4-1. Set up a debug configuration for a specific application project.

4-1-1. Right-click the application project that you want to build the Debug configuration for from the Project Explorer pane (1).

4-1-2. Select **Debug As** to open the menu of predefined configurations and the configuration manager (2).

4-1-3. Select **Debug Configurations** to view all the available debug options (3).

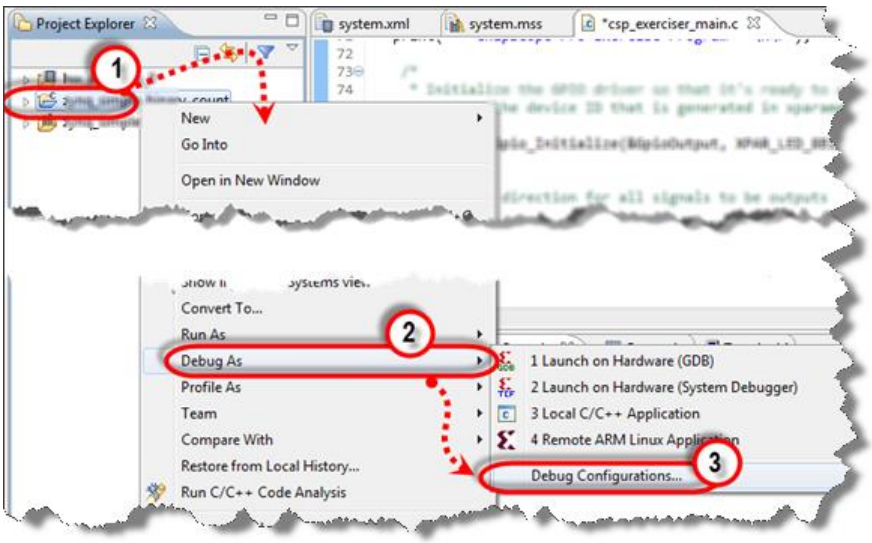


Figure 11-11: Creating a Debug Configuration

The Debug Configurations dialog box opens.

4-1-4. Select **Xilinx C/C++ application (System Debugger)** since you will be debugging this type of system (1).

GDB still works; however, it is considered deprecated for new designs.

4-1-5. Click the **Create New Configuration** icon to create the new configuration for your application (2).

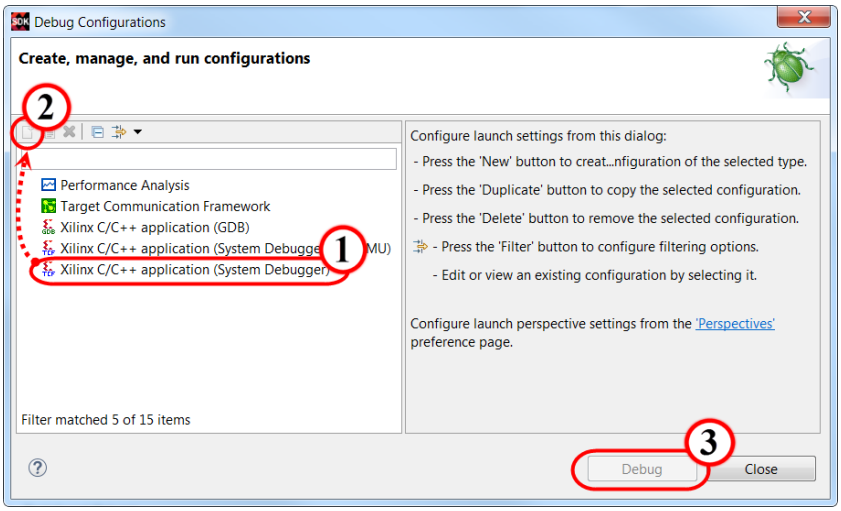


Figure 11-12: Creating a New Debug Configuration

A new Debug Configuration is created. The previous figure depicts an SDK workspace that does not yet have any debug or run configurations defined. If other configurations did exist, or after the creation of a first configuration, the new configuration will appear under the type of configuration.

Note the default configuration name and other parameters that are automatically filled in. In most cases, you just need to click the Debug button to begin the session. This Debug Configuration menu is useful when you want to later change debug parameters.

The new configuration will appear with other existing configurations and have the name of your application. You will also note that a number of fields are automatically filled in for you using the name of your application as the basis (that is, if your application is named "XYZ", then the Name field will be populated with XYZ Debug, and the C/C++ Application field will be populated with Debug\XYZ.elf).

4-1-6. Click **Debug** to close the window and launch the debugging session (3).

If the Confirm Perspective Switch dialog box appears, click **Yes**.

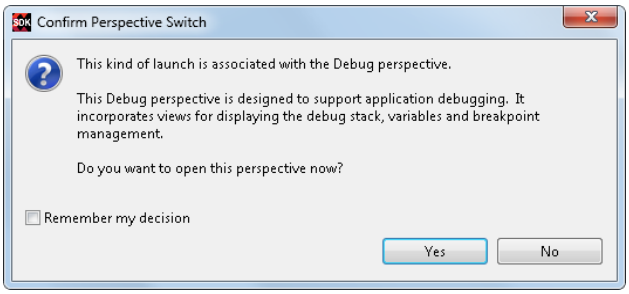


Figure 11-13: Confirming Switch of Perspective

4-1-7. The Debug Perspective view opens.

For Linux users, proceed with the instructions below.

Run and Debug configurations are application project objects that contain communication, hardware, and execution options for running an application on a hardware or emulation platform. The selections for *Run* and *Debug* are identical and only differ in that *Run* just executes the application and *Debug* opens a debug perspective and launches a debug program.

There are different types of Run and Debug configurations based on the operating system (or Standalone libraries) and the SDK download/debug tools that you want to use. Multiple configurations can be defined for any project. This facilitates enhanced development and debugging.

4-2. Create a new Linux Debug configuration. Debug and Run configurations associate an ELF object file to a target (typically a hardware board) for execution. In this case, the target is a hardware board accessed over the Ethernet TCP/IP connection.

4-2-1. Click the **C/C++** tab in the upper-right corner of the GUI to return to the C/C++ perspective.

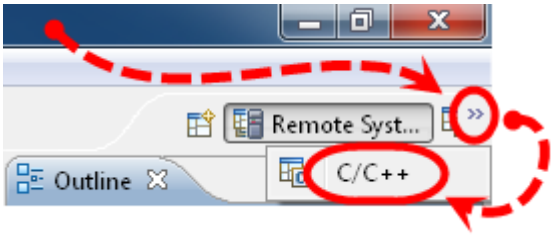


Figure 11-14: Changing Perspective

This brings back access to the software projects. It is accessed by first clicking the **>>** in the same location. If this tab is not available, you can also return to the perspective by selecting **Window** > **Open Perspective** > **Other** > **C/C++ (default)**.

4-2-2. Right-click **profLab** in the Project Explorer window (1).

4-2-3. Select **Debug As** (2) > **Debug Configurations** (3).

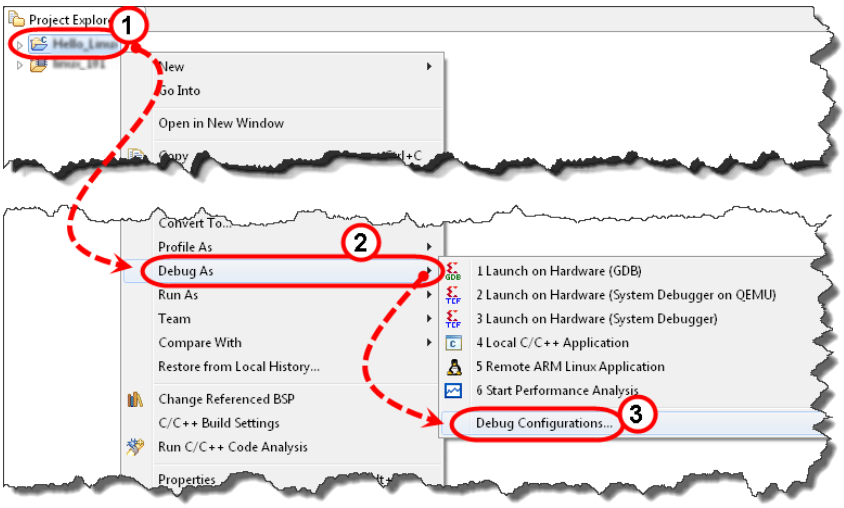


Figure 11-15: Selecting Debug Configurations

The Debug Configurations dialog box opens.

4-2-4. Double-click **Xilinx C/C++ application (System Debugger)** to create a launch configuration (1).

Alternatively, you can select **Xilinx C/C++ application (System Debugger)** and click the **New** configuration button.

If this is the first debug configuration being created for the project, a welcome type dialog opens. If one or more configurations exist, then the last open configuration will be displayed. In either case, a new configuration can always be added. Existing configurations are shown in the left pane and can be selected for debugging.

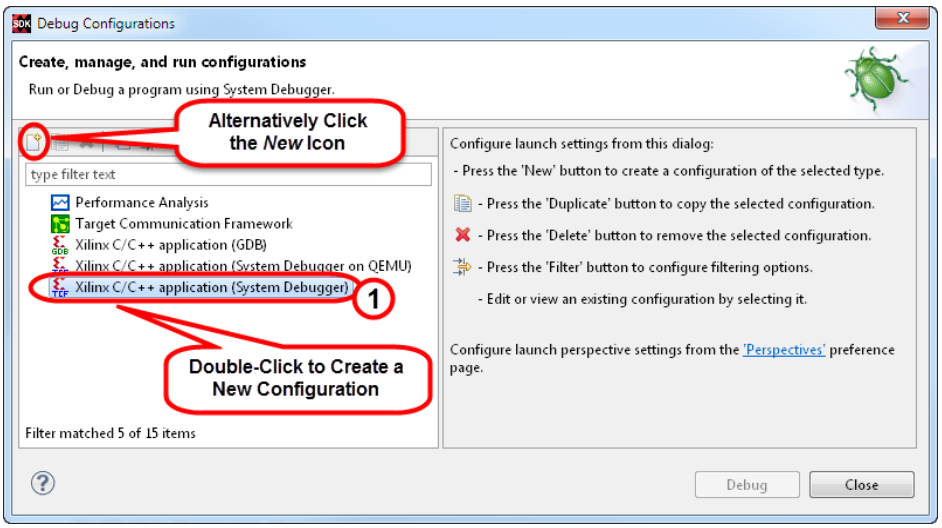


Figure 11-16: Creating a New System Run/Debug Configuration

4-3. Configure the debug type and host connection.

4-3-1. Select **Linux Application Debug** from the Debug Type drop-down list (2).

This will engage the proper debug tool to use.

4-3-2. Click **New** next to the Connection drop-down list to launch the Target Connection Details dialog box (3).

A new connection will be defined from the debugger to the hardware (or emulator) target.

4-3-3. Enter **ZynqBoard** in the Target Name field as the name of the connection (4).

4-3-4. Enter **192.168.1.10** in the Host field (5).

This is the IP address of the RSE connection that was set up earlier.

4-3-5. Enter **1534** in the the Port field (5).

This is the default TCP/IP port number for the connection.

4-3-6. Click **OK** (6).

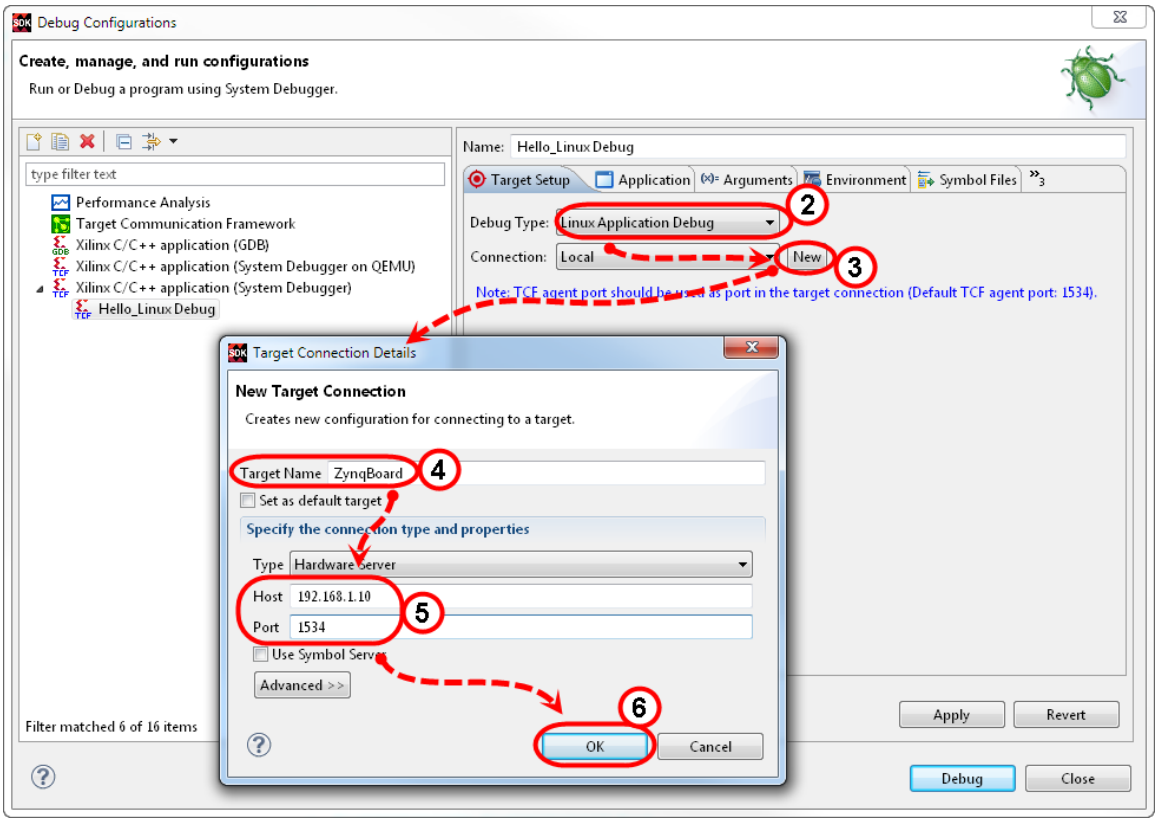


Figure 11-17: Selecting the Debug Type and Connection

4-4. Select the software application ELF file to debug and its file path on the remote target board where it is to be copied.

The actual software application debugging takes place on the Linux platform running on the target hardware, so it is necessary to put a copy of the ELF file on it.

4-4-1. Select the **Application** tab (1).

This is where the software application is selected and allows configuration of where the application is placed in the remote file system.

4-4-2. Click **Browse** next to the Local File Path field to select the software application ELF file (2).

4-4-3. Navigate to and select the file **C:\training\SDK_Profiling\lab\profLab\Debug\profLab.elf** (3).

4-4-4. Enter **/tmp/profLab.elf** in the Remote File Path field as the location on the target platform where the application ELF file will be copied (4).

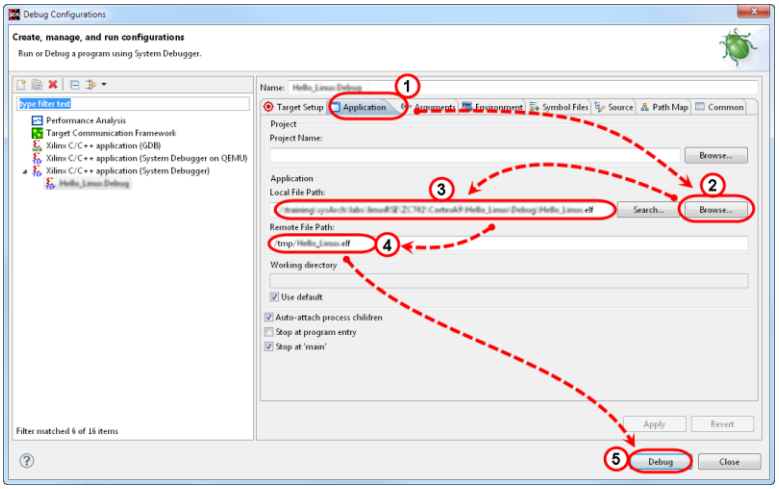


Figure 11-18: Selecting the Local File Path and Remote File Path

The remaining options will be accepted at their default values.

4-4-5. Click **Debug** (5).

4-4-6. Click **Yes** to confirm opening the Debug perspective view.

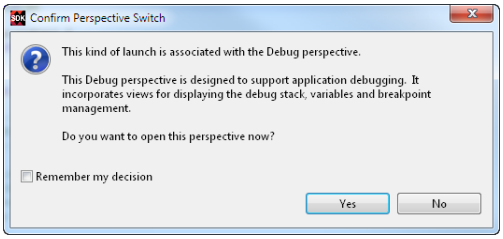


Figure 11-19: Confirming Perspective Switch

The Debug perspective opens.

4-5. Optional for Linux: Highlight the process for debug.

Sometimes the process for debugging is not highlighted in the Debug view and the flow controls such as pause, resume, single step, etc. will be grayed out. Follow the procedure below to enable the controls.

4-5-1. Click the down arrow in the Debug view, if it is not already expanded, to expand the top-level debug session (1).

4-5-2. Click the down arrow, if it is not already expanded, to expand the next level showing the binary that is being executed (2).

Note: The location of the process on the remote device (**/tmp/profLab.elf**) is also visible.

4-5-3. Click the process that is currently suspended (3).

Note: This will tell the Debug tools which remote application/process to debug and enable the flow control tools.

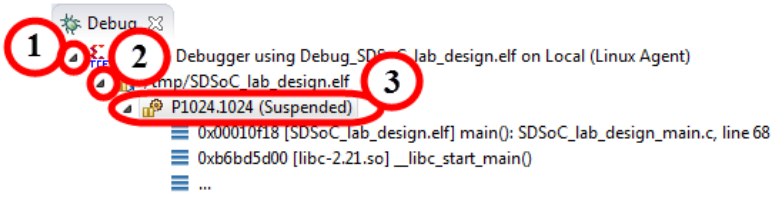


Figure 11-20: Highlighting the Debug Session Application

Tip: In the figure above and below, P1024 is the process ID (PID) of the application running under Linux; your PID may be different. Issuing the `ps` command through the terminal will show all of the system processes that are currently running. There will be a listing with the same PID that matches the number of your suspended application.

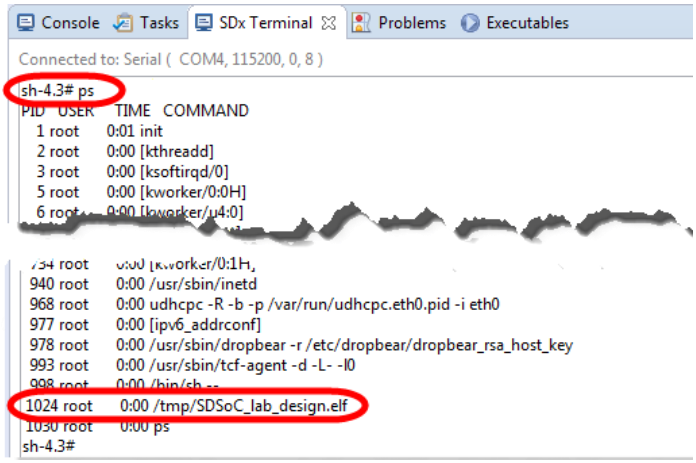


Figure 11-21: Linux Terminal – ps Command

You will now set up and run profiling for the project.

4-6. Open the TCF Profiler to view where the processor is spending time in the code.

4-6-1. Select **Window** (1) > **Show View** (2) > **Other** (3).

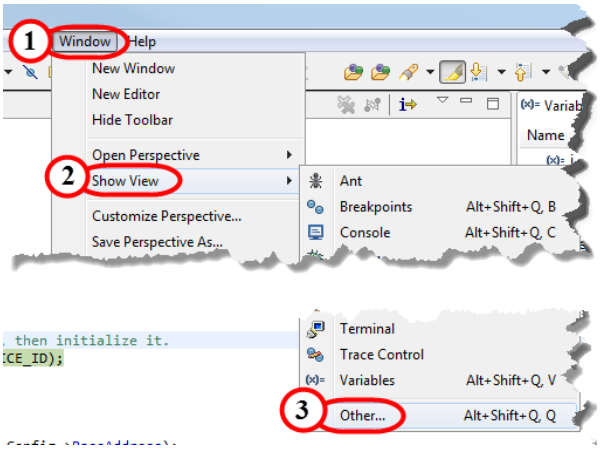


Figure 11-22: Viewing Hotspots

The Show View dialog box opens.

4-6-2. Expand **Debug** and select **TCF Profiler**.

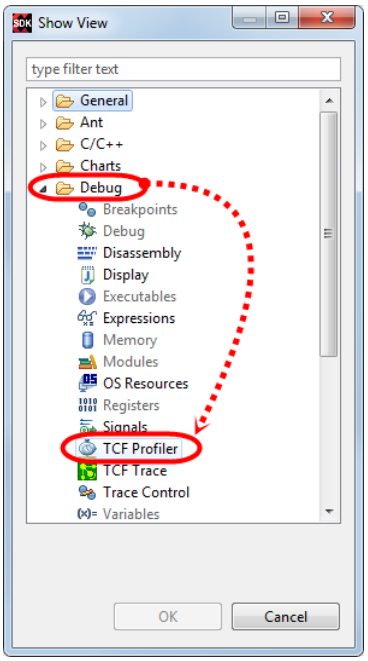


Figure 11-23: Selecting TCF Profiler

4-6-3. Click **OK**.

This opens the TCF Profiler view in the SDK tool.

4-6-4. Click **Start** in the TCF Profiler view in the tool on the right side.

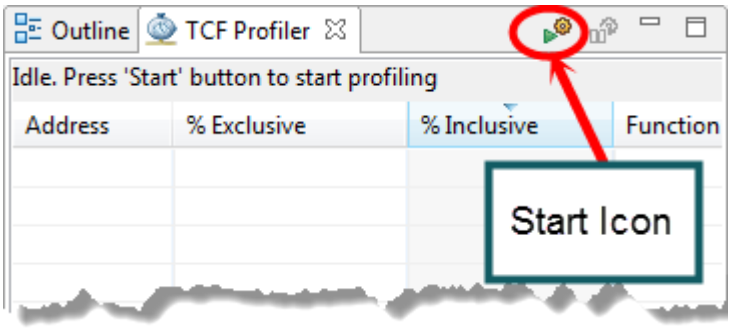


Figure 11-24: Starting the TCF Profiler

4-6-5. Select **Aggregate per function** in the Profiler Configuration dialog box.

4-6-6. Select the **Enable stack tracing** option if it is not selected.

The update rate refers to how often the information on the screen is updated. The default setting is four seconds.

The *Enable stack tracing* and *Max stack frames count* options are useful for debugging stack overflows.

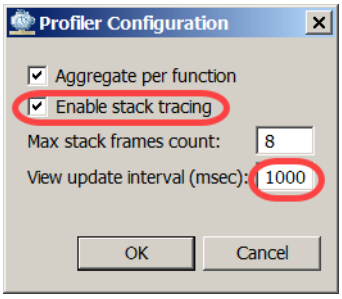


Figure 11-25: Selecting the Profiler Configuration

4-6-7. Click **OK** to accept these settings and arm the profiler.

4-7. Run the software application.

4-7-1. Click the **Resume** icon to run the program.

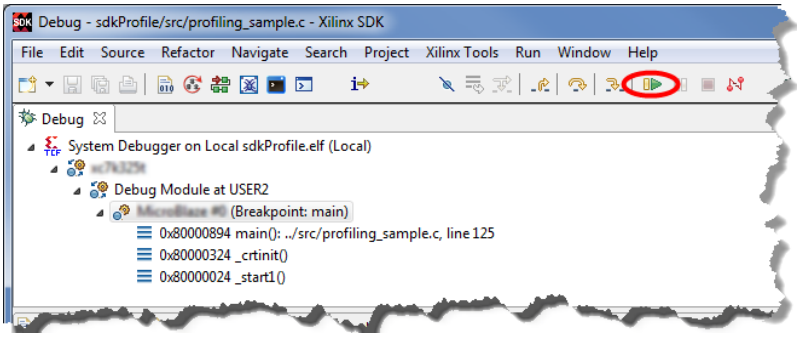


Figure 11-26: Resuming the Program

The program will stop at the previously enabled breakpoint in the *main* function.

4-7-2. Select the **TCF Profiler** window.

Note: You can drag this tab to the main view or resize the current pane to see the contents more clearly.

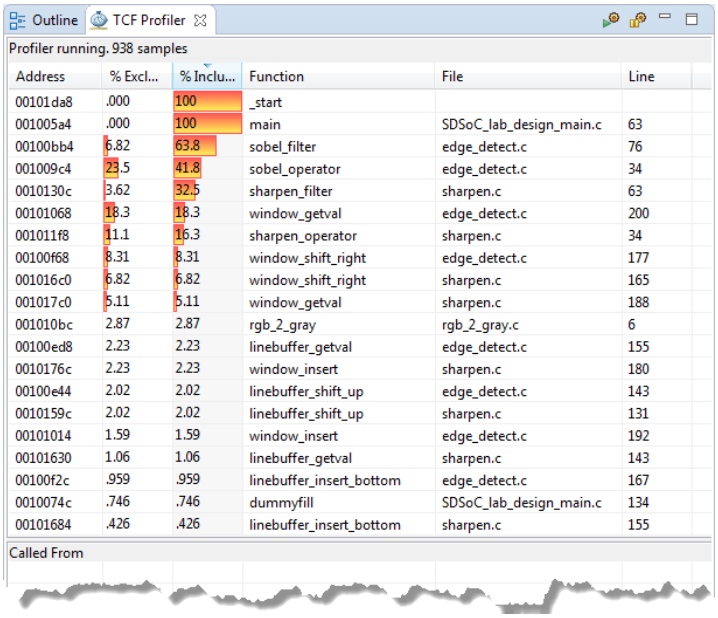


Figure 11-27: Profiling Sample Results

Note: You may see slightly different values due to the sampling.

Address is the location of the function in the Disassembly view (i.e., its location in memory).

% Exclusive is the percentage of samples encountered by the profiler for that function only (does not take into consideration samples of child functions). This can also be seen as the exclusive percentage for that particular function. This parameter can be of help in identifying performance bottlenecks.

% Inclusive is the percentage of samples of a function, including samples collected during execution of child functions.

Function is the function being sampled.

File is the file containing the function definition.

Line is the line where the function is defined in the said file.

4-7-3. Click the function name **sobel_filter** in the TCF Profiler window.

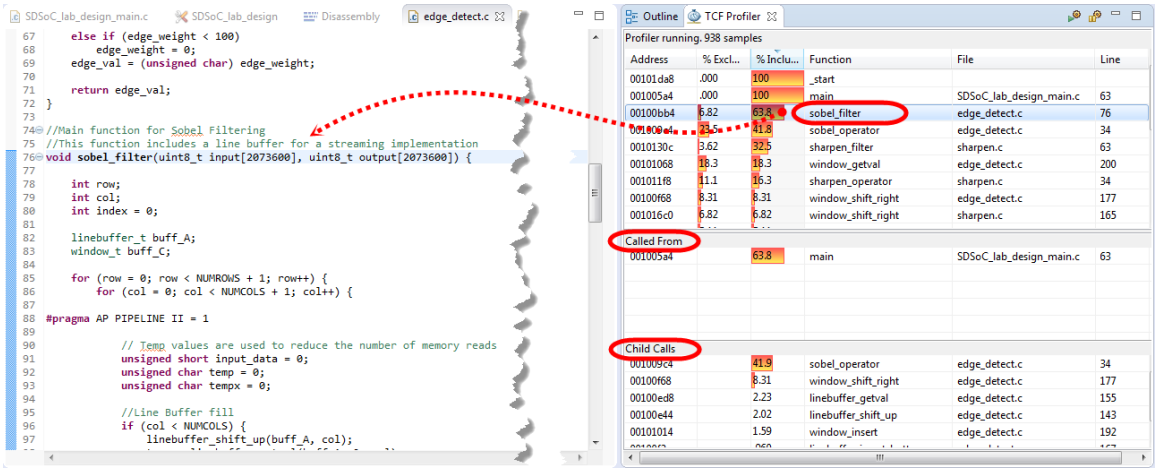


Figure 11-28: Viewing the Sobel Filter from the Profiler

This will open the *edge_detect.c* file in the editor and the selected function will be highlighted.

Selecting the function in the TCF Profiler can also show where it is called from and who its child calls are.

Question 1

What are the child functions of *main*?

Question 2

What are the parent and child functions of *sobel_operator*?

Question 3

How many times are x_weight and y_weight calculated every time *sobel_operator* is called?

Question 4

How many times will *sobel_operator* be called for the image of size 1920 * 1080?

Question 5

How many times will x_weight and y_weight be calculated for each image passed through *sobel_filter*? **Hint:** Review the answers to the previous two questions.

Question 6

What does the program *main()* do?

Question 7

What are the % Exclusive values for the functions *sobel_filter()*, *sharpen_filter()*, and *rgb_2_gray()*?

Analyzing the TCF Data Step 5

Using the results from the TCF Profiler, you will analyze the function's exclusive and inclusive values, determine which uses the most execution time, and trace the program hierarchy.

The following diagram provides a visual representation of exclusive and inclusive.

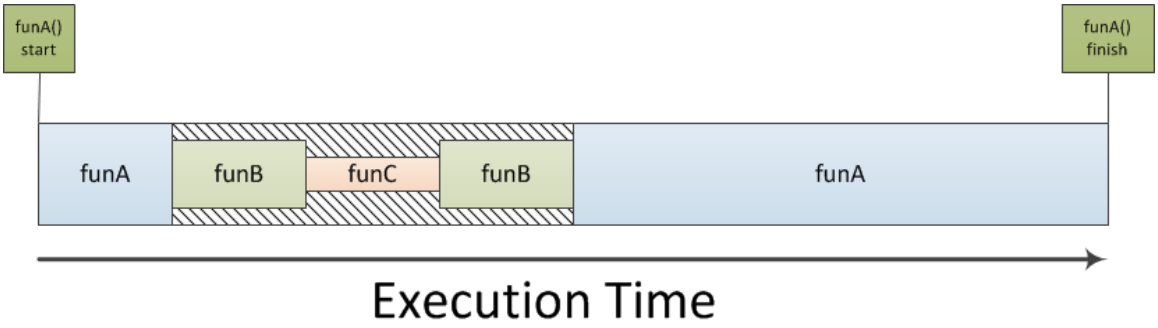


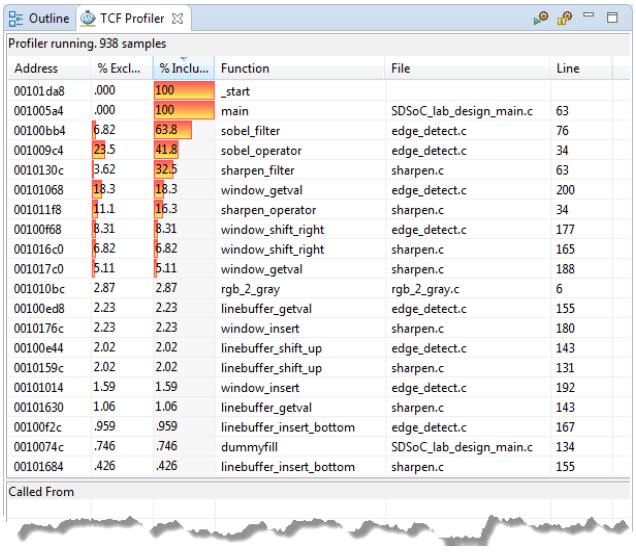
Figure 11-29: Exclusive vs Inclusive Pipelined

Exclusive: The amount of execution time spent in *funA* alone. Referencing the above diagram, the exclusive time for *funA* is represented by the combined execution time of the funA blocks only.

Inclusive: The amount of execution time spent in *funA* and all of its sub-function calls. From the diagram, this is the exclusive time of funA combined with the hatched area during which time *funB* and *funC* are executing.

5-1. View the sample results.

5-1-1. Open the **TCF Profiler** viewer with your profile results.


Outline | TCF Profiler

Profiler running. 938 samples

Address	% Excl...	% Inclu...	Function	File	Line
00101da8	.000	100	_start		
001005a4	.000	100	main	SDSoC_lab_design_main.c	63
00100bb4	6.82	63.8	sobel_filter	edge_detect.c	76
001009c4	23.5	41.8	sobel_operator	edge_detect.c	34
0010130c	3.62	32.5	sharpen_filter	sharpen.c	63
00101068	18.3	18.3	window_getval	edge_detect.c	200
001011f8	11.1	16.3	sharpen_operator	sharpen.c	34
00100f68	8.31	8.31	window_shift_right	edge_detect.c	177
001016c0	6.82	6.82	window_shift_right	sharpen.c	165
001017c0	5.11	5.11	window_getval	sharpen.c	188
001010bc	2.87	2.87	rgb_2_gray	rgb_2_gray.c	6
00100ed8	2.23	2.23	linebuffer_getval	edge_detect.c	155
0010176c	2.23	2.23	window_insert	sharpen.c	180
00100e44	2.02	2.02	linebuffer_shift_up	edge_detect.c	143
0010159c	2.02	2.02	linebuffer_shift_up	sharpen.c	131
00101014	1.59	1.59	window_insert	edge_detect.c	192
00101630	1.06	1.06	linebuffer_getval	sharpen.c	143
00100f2c	.959	.959	linebuffer_insert_bottom	edge_detect.c	167
0010074c	.746	.746	dummyfill	SDSoC_lab_design_main.c	134
00101684	.426	.426	linebuffer_insert_bottom	sharpen.c	155

Called From

Figure 11-30: Profiling Sample Results

Note: The values in the figure below came from a separate run of the TCF Profiler than those in the screenshot above. As the Profiling results are probabilistic, variation in results between runs is expected. Your results will not exactly match those provided in the figure and diagram.

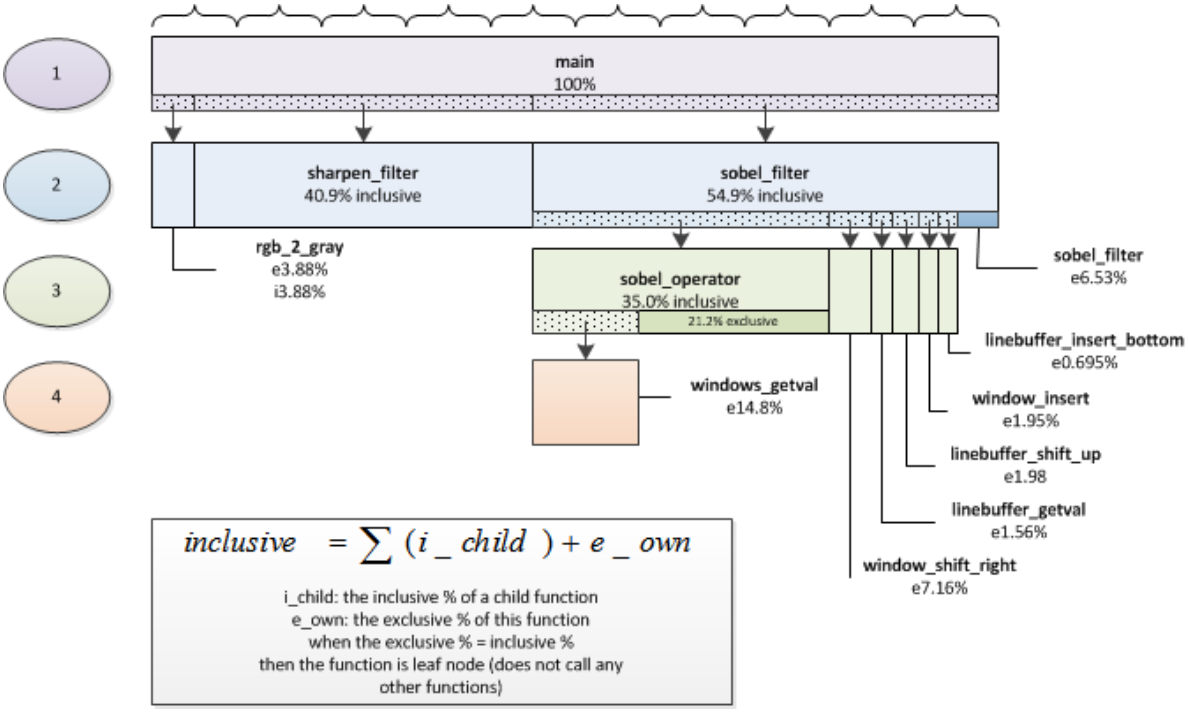


Figure 11-31: Visual Analysis of TCF Profiler

In the figure above a detailed hierarchy of the function *sobel_filter* is shown. Focusing on the *sobel_filter* hierarchy, note that there are four levels of function calls. The root function is *main,* which runs from level 1. As all other functions are called from *main*, *main* takes up 100% of the inclusive execution time, yet ~ 0% of the exclusive execution time. The function *main* calls *sobel_filter*, *sharpen_filter*, and *rgb_2_gray*, all level 2 functions, which take up 54.9%, 40.9%, and 3.88% of the execution time, respectively.

For this illustration, you are focusing on the *sobel_filter* function. The exclusive value is reported to be 6.53%. Referring to the diagram above, this means that *sobel_filter* calls several level 3 functions that take up (54.9% - 6.53%) = 48.37% of the execution time. All of the level 3 functions besides *sobel_operator* are leaf nodes (their exclusive value equals their inclusive value) and they make no calls to any other functions.

sobel_operator is a level three function that takes up 35.0% of the execution time inclusive and 21.2% exclusive. (35.0 - 21.2) = 14.8% of the execution time accounted for by *sobel_operator* are from level 4 functions.

Note: The equation provided in the above diagram may be of interest when you are working out the inclusive value of any given function. It is a recursive equation where the base case occurs at the leaf nodes.

Note: All exclusive percentages are out of the total 100% regardless of the level of the function. For example, *windows_getval* is reported to have an exclusive % of 6.97. This is 6.97% of 100% from *main* and not 6.97% of the 35.0% from its parent function *sobel_operator*.

Note: Due to sampling differences and binning errors, there may be some deviance in the numbers.

Based on the above diagram, drawn from the TCF Profiler results, it can be seen that the level 3 function *sobel_operator* takes up 21.2% of the execution time. Also the functions *windows_getval* and *window_shift_right* take up ~15% and ~7% each. Combined, this is ~43% of the total execution time of the system.

As the function *sobel_filter* is concerned only with edge detection, and three components of this image process take up ~43% of the execution time, these functions make good candidates to analyze for acceleration.

5-2. Analyze the *sharpen_filter* hierarchy.

The purpose of this instruction is for you to inspect the *sharpen_filter* function, fill out the diagram below and answer some questions on the information collected. The TCF Profiler results provides all of the required information to be able to fill out the diagram below.

You can reference the diagram for the *sobel_operator* above for a template on how to fill out the values. The inclusive and exclusive values for the sharpen filter have been provided as a starting point.

Question 8

Fill out the hierarchy diagram below.

- Insert all of the function names.

Tip: Do not get caught up matching exclusive % values to block sizes.

- Insert the exclusive percentages for all functions by filling in the e____% fields underneath the function names filled in from the previous task.
- Insert the inclusive percentage for *sobel_operator*.

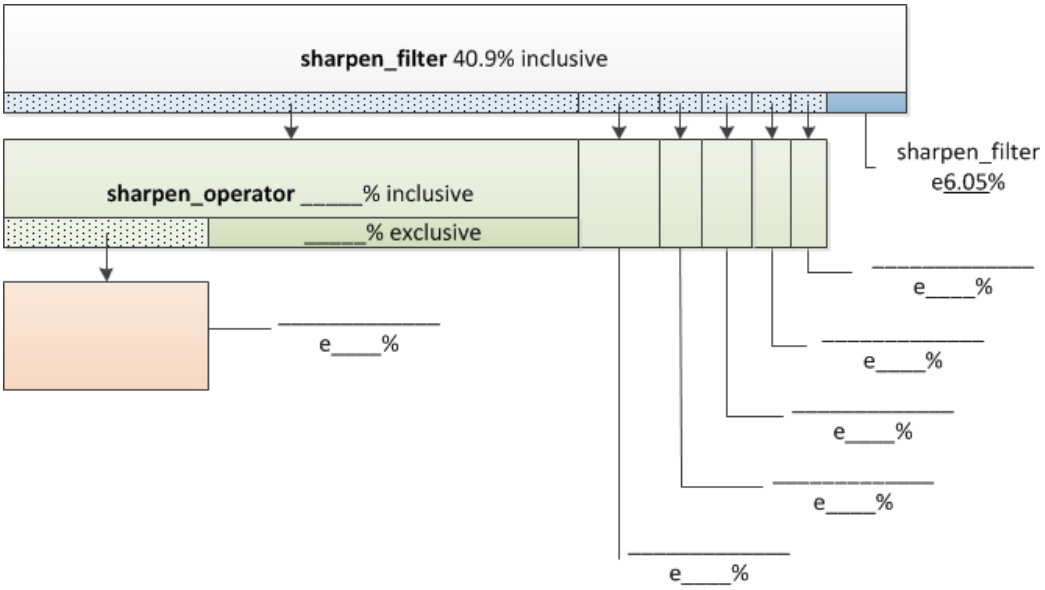


Figure 11-32: Sharpen Filter Exercise

Question 9

What function is the next largest consumer of the exclusive execution time?

Question 10

What is the difference between exclusive and inclusive?

Question 11

Why do all of the inclusive values shown in your results add up to more than 100% and all of the exclusive values add up to 100%?

Question 12

Why are there not any MACROs shown in the TCF Profiler results? (MACROs are used frequently in this project.)

Question 13

Why are the exclusive and inclusive values of the function *rgb_2_gray* equal?

Summary

You profiled several software algorithms and performed a more detailed analysis of one of these that may be a candidate for hardware implementation.

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. What are the child functions of *main*?

   sobel_filter, *sharpen_filter*, *rgb_2_gray*, xlnkAllocBufInternal (if running Linux) and *dummyFill*.

2. What are the parent and child functions of *sobel_operator*?

   The parent is *sobel_filter* and the only child is *window_getval*.

3. How many times are x_weight and y_weight calculated every time *sobel_operator* is called?

   Nine times. The WINDOW_HEIGHT and WINDOW_WIDTH are each 3 respectively, resulting in 3*3 = 9.

4. How many times will *sobel_operator* be called for the image of size 1920 * 1080?

   sobel_operator will be called 1920 * 1080 times because it must perform its calculation for each and every pixel in the image. This is 2,073,600 times that the function is called.

5. How many times will x_weight and y_weight be calculated for each image passed through *sobel_filter*? **Hint:** Review the answers to the previous two questions.

   x_weight and y_weight are each calculated (2,073,600 * 9) = 18,662,400 times. Combined, this is nearly 40 million operations that this function alone needs for an image of 1920 * 1080.

6. What does the program *main()* do?

   It creates image buffers, then fills the color image with data, converts this to grayscale, and performs a sharpen and edge detect on the grayscale image. This is a computationally taxing sequence of processes, especially for large images.

7. What are the % Exclusive values for the functions *sobel_filter()*, *sharpen_filter()*, and *rgb2gray()*? **Note:** Your values may differ.

   - *sobel_filter()* ~6.24%
   - *sharpen_filter()* ~3.42%
   - *rgb2gray()* ~3.42%

8. Fill out the hierarchy diagram below. **Note:** Your values may differ.

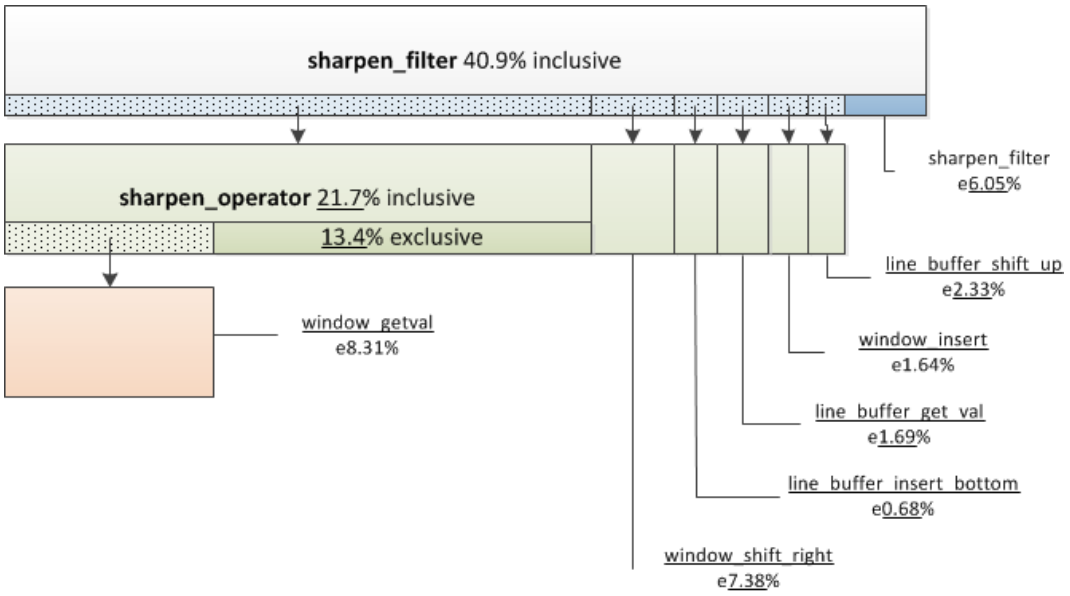


Figure 11-33: Sharpen Filter Exercise - Completed

9. What function is the next largest consumer of the exclusive execution time?

   sharpen_operator utilizes ~13 exclusive and ~ 21% inclusive.

10. What is the difference between exclusive and inclusive?

    Exclusive only considers the execution time for a given function in its own level. Inclusive considers the execution time of a function and all of its child function calls.

11. Why do all of the inclusive values shown in your results add up to more than 100% and all of the exclusive values add up to 100%?

    The exclusive values represent the actual time taken to run the function. As the execution times of all functions in an application add up to the total execution time of an application, this is 100%. The inclusive values take into account the time taken to run that function and all child function calls. When the inclusive values are added together, the execution times of lower level functions are added multiple times, resulting in a value that is greater than 100%.

12. Why are there not any MACROs shown in the TCF Profiler results? (MACROs are used frequently in this project.)

    MACROs are unrolled and inserted into the code by the preprocessor—meaning that wherever a MACRO is called, its actual function is determined during preprocessing and inserted into the code in place of the MACRO call. A function call, unless it is forced inline, will be a separate call that the TCF profiler can analyze.

13. Why are the exclusive and inclusive values of the function rgb_2_gray equal?

    This is a leaf node and calls no other functions.

Lab 12: Writing a Device Driver

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

This lab demonstrates how to create a skeleton driver framework and add a provided LCD device driver to the BSP.

Objectives

After completing this lab, you will be able to:

- Create a device driver skeleton by using the Create and Package IP Wizard
- Edit the **.mdd* file to attach to a peripheral
- Write code to replace the driver for the *axi_gpio* peripheral
- Successfully include the device driver in the BSP
- Call the BSP services from a software application

Introduction

A device driver is a lower-level service that typically interfaces the software application with a specific piece of hardware. Device drivers can either be part of your software application project or part of the board support package (BSP). When it is part of your application project, the source code for the drivers must always be present, compiled, and linked with your application. If the driver is used often and is stable, a better alternative is to make it a service of the BSP. This makes it easier to distribute and the programmer does not have to be concerned with keeping track of it in the application project.

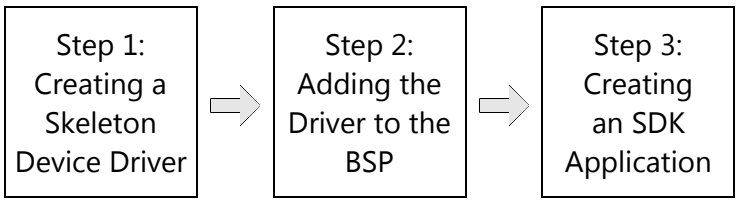
All Xilinx-provided peripherals have device drivers that are included in the BSP when they are used. When a custom peripheral is created via the Create and Package IP Wizard, a skeleton software driver for BSP inclusion is provided, along with a hardware skeleton.

Because this lab is a software exercise and there is limited time, you will not be able to build a custom piece of hardware and a software driver for it. Instead, you will create a custom driver for the *axi_gpio* bus peripheral. The process will illustrate the ability of SDK to support multiple software drivers for any given hardware peripheral.

The hardware platform in this class uses an *axi_gpio* peripheral as a controller for the LCD display on the Zynq® All Programmable SoC ZC702 or Zed board. The timer software application project in the "Application Development" lab includes a low-level driver for the LCD display.

The structure of the device driver environment is sensitive to directory/file names and locations. Generating **.mdd* and **.tcl* definition files will be necessary. To make the task easier, you will use the Create and Package IP Wizard in the Vivado® Design Suite to generate dummy hardware peripheral and device driver skeletons. You will ignore the hardware portion and build on the software structure that the wizard generates.

General Flow



Creating a Skeleton Device Driver — Step 1

The first step of this lab will be to create an LCD dummy peripheral using the Create and Package IP Wizard in the Vivado Design Suite to generate the skeleton structure of the device driver.

There are a number of ways to launch the Vivado Design Suite. The two most popular mechanisms are shown here.

1-1. Launch the Vivado Design Suite.

This can be done in two standard ways, use your preferred method.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2016.3** > **Vivado 2016.3**.

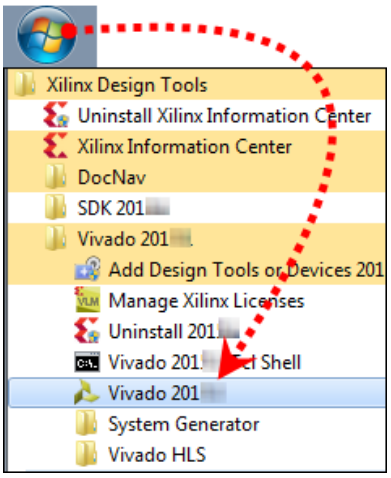


Figure 12-1: Launching the Vivado Design Suite from the Start Menu

-- OR --

Double-click the **Vivado Design Suite** shortcut icon ( ) on the desktop.

The Vivado Design Suite opens to the Welcome window. From the Welcome window you can create a new project, open an existing project, or enter Tcl commands directly into the Vivado Design Suite as well as access documentation and examples.

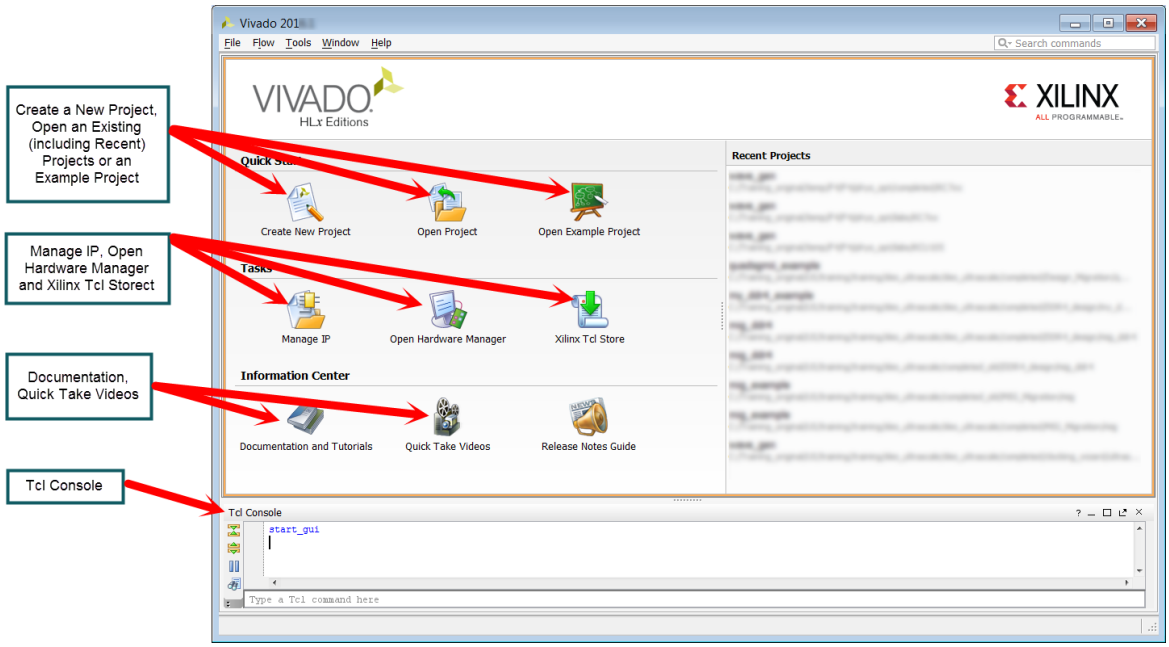


Figure 12-2: Vivado Design Suite Welcome Screen

With the Vivado Design Suite now open, you will load a helper Tcl script and run it to configure the project and get you to the important part of this lab—working with the processor.

The Vivado Design Suite offers both GUI and scripted control. Scripted control takes the form of Tcl commands. These Tcl commands can be entered directly into the tool one at a time, or an entire Tcl script can be loaded and executed.

1-2. Run a Tcl script.

1-2-1. Locate the Tcl command line entry.

The command line entry can be found either on the Welcome page prior to a project being opened, or once a project has been opened.

From the Welcome screen:

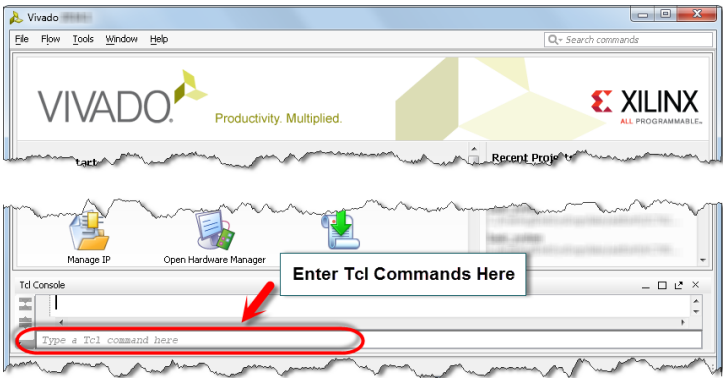


Figure 12-3: Accessing the Tcl Console from the Getting Started Page

From an opened project:

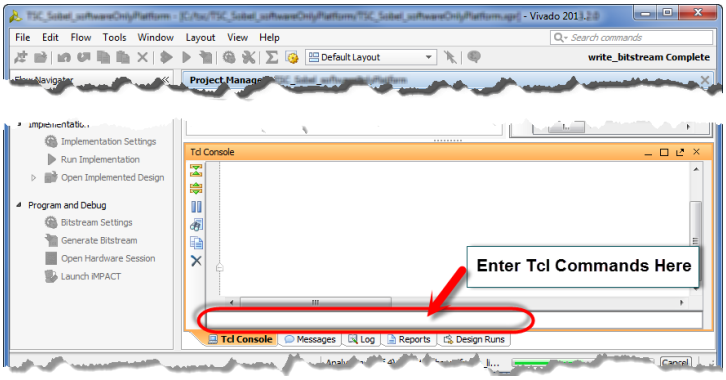


Figure 12-4: Entering Commands into the Tcl Console from an Open Project

The default directory for the Tcl environment is nested within the Xilinx installation directory. This placement, however, is often disadvantageous. In most cases, you will want to navigate to a more useful path. To do this, use the `cd` command to change directory to the user directory.

1-2-2. Change the current working directory to where the Tcl script is located by entering:

```
cd C:/training/deviceDriversCustom/support
```

Remember that the Tcl environment is based on Linux and requires the '/' character to delimit hierarchical paths.

1-2-3. Verify that you are now where you want to be by entering the following into the Tcl command line:

```
pwd
```

The current working directory is displayed. If you are not where you want to be, use the `cd` command to change to `C:/training/deviceDriversCustom/support`.

1-2-4. Enter the following Tcl command:

```
source deviceDriversCustom_completer.tcl
```

The Tcl script is run as though you typed each command included in the Tcl script into the Tcl command line. You can follow the execution of the script and monitor for any errors or warnings in the Tcl Console.

With the Tcl script now loaded, you will run the *createProject* proc to build the Vivado Design Suite project and create an appropriately named block design in which you will perform the remainder of this lab.

1-3. Run the *createProject* proc to quickly create the Vivado Design Suite Project.

1-3-1. Enter the following into the Tcl command line to create the Vivado Design Suite project:

```
createProject
```

1-4. Launch the Create and Package IP Wizard, which will build a framework for the custom device driver.

1-4-1. Select **Tools** > **Create and Package IP** to start the wizard.

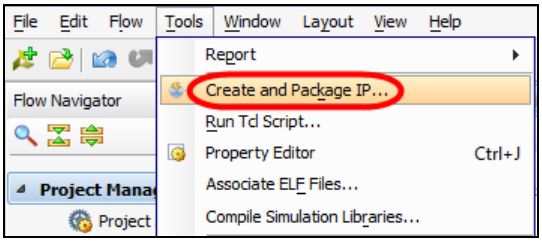


Figure 12-5: Selecting Create and Package IP

The Welcome dialog box for the Create and Package IP Wizard opens.

Figure 12-6: Viewing the Welcome Dialog Box

This wizard is used to create and package a custom hardware peripheral. At completion, it generates a skeleton software driver. Because this lab is not concerned with the hardware content that consumes most of the wizard choices, you will select the default settings for the hardware choices. The second-to-last dialog box of the wizard presents the option to generate a skeleton device driver structure.

1-4-2. Click **Next** to add a peripheral to the project.

1-5. Create templates for a new AXI4 peripheral. Indicate where you want to store the new peripheral.

1-5-1. Select **Create a new AXI4 peripheral** when the Create And Package New IP dialog box opens.

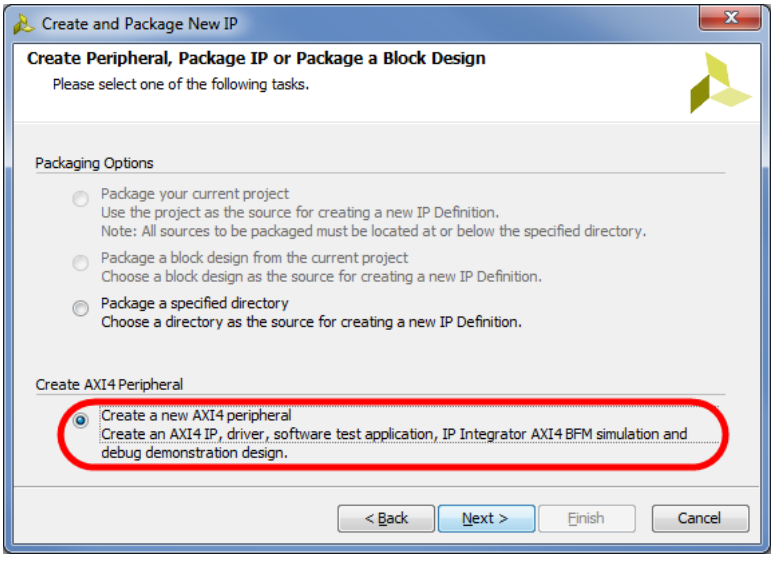


Figure 12-7: Selecting AXI4 Peripheral

1-5-2. Click **Next** to enter initial peripheral settings.

1-6. Enter the name of the peripheral.

1-6-1. Enter **lcd** in the Name field.

1-6-2. Change the IP location to **C:\training\deviceDriversCustom\lab**.

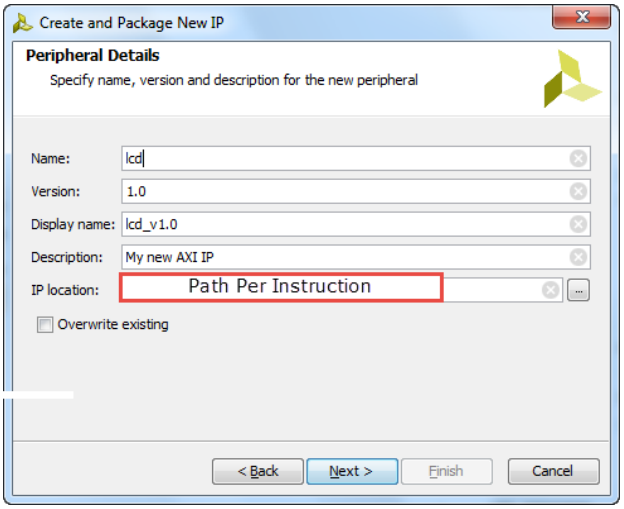


Figure 12-8: Naming the Peripheral and IP Location

1-6-3. Click **Next** to add peripheral settings.

1-7. Add a single AXI interface.

1-7-1. Review the default settings in the Add Interface dialog box.

No changes are required.

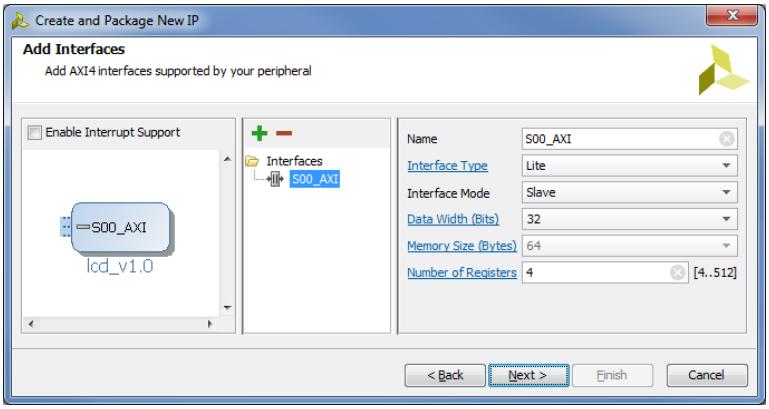


Figure 12-9: Adding a Single Interface

1-7-2. Click **Next** to review your previous entries.

1-8. Review the summary of the peripheral that will be created.

1-8-1. Ensure that the **Add IP to the repository** option is selected so that you will be able to manually modify the device driver.

1-8-2. Click **Finish** to add the peripheral to the project.

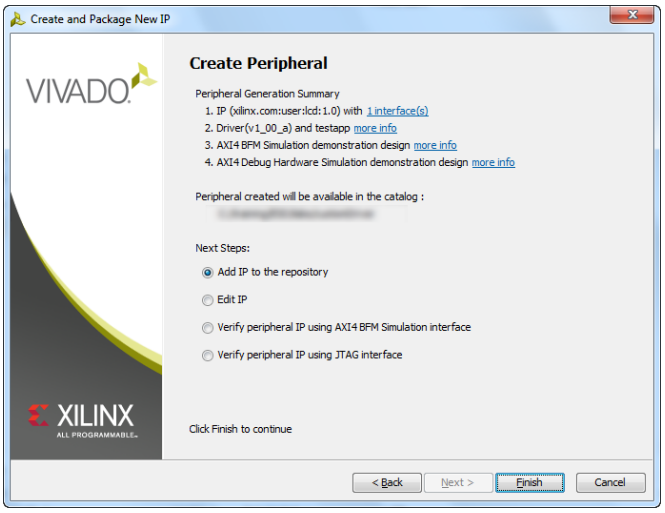


Figure 12-10: Reviewing the Summary

No simulation files are required. This dialog box is the only one that concerns the *lcd* peripheral. Although minimum custom hardware is actually being made, it will not be used in this lab. The objective is to add a custom driver to an existing peripheral. To create a custom driver, the template driver files are necessary.

1-9. View the directory structure created for the driver peripheral.

1-9-1. Browse to *C:\training\deviceDriversCustom\lab* using Windows Explorer.

The Create and Package Wizard has added the new *drivers* directory to the Vivado IDE project structure.

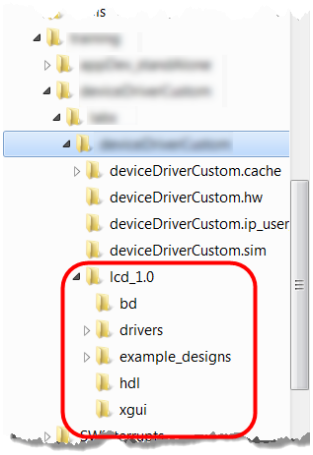


Figure 12-11: Project Hierarchy with Results of Create and Package Wizard

Question 1

Browse to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory. This directory is where the source files for the device driver reside. What files have been placed there by the wizard?

Question 2

What is in the *lcd.c* file?

No hardware was changed or created so the design does not have to be reimplemented. The goal of the process is to associate a custom driver to existing peripheral hardware. The new peripheral is named *lcd* but the hardware is actually an unmodified *gpio* peripheral.

1-9-2. Select **File** > **Exit** to close the Vivado Design Suite now that the hardware aspect is done.

Adding the Device Driver to the BSP Step 2

The skeleton driver that was added in the last step needs to be customized as an *axi_gpio* peripheral driver. You will accomplish this by modifying the *lcd.mdd* device driver description file.

The *lcd.c* device driver file from the "Application Development" lab will replace the skeleton version of the same name that was generated by the wizard. The generated *lcd.h* file will not be needed because the drivers in *lcd_zynq.c* will actually be calling the *axi_gpio* drivers that are already in the BSP.

This is a valuable feature in that device drivers in the BSP (or software platform) can call other device driver services in the same BSP.

Adding the device driver to the BSP requires you to:

- Modify *lcd.mdd* to point to the *axi_gpio* peripheral by using SDK.

- Replace the *lcd.c* file located in the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory with the custom driver files from the *C:\training\deviceDriversCustom\support* directory.
- Add the path for the custom *lcd* driver files to the user-defined software repositories.
- Create a new BSP and modify its settings to use the *lcd* driver for the LCD display.
- Regenerate the BSP.

The first step will have you modify the configuration files that integrate the driver source into the BSP. They are in the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\data* directory.

The *lcd.mdd* file needs to be modified so that this software driver is associated with the *axi_gpio* peripheral. The *lcd.tcl* file does not need modification because it generates symbols that will not be used in the *xparameters.h* file.

2-1. Launch the SDK tool and set the workspace.

2-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

2-1-2. Enter **C:\training\deviceDriversCustom\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\deviceDriversCustom\lab** directory and click **OK**.

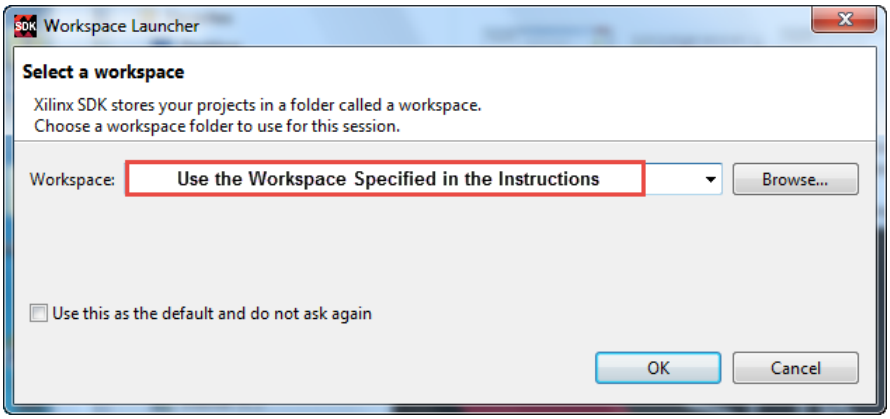


Figure 12-12: Setting Up the Workspace Environment Path

2-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

2-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

The hardware platform specification contains a thorough description of the hardware design: what types of processors are present, active peripherals in the PS and PL for Zynq® All Programmable SoC-based systems or a list of all peripherals for a non-Zynq All Programmable SoC system, a full system memory map, etc. Based on this description, software such as the board support package (BSP) and application can be tailored to the hardware.

2-2. Create the hardware platform specification.

2-2-1. Select **File** (1) > **New** (2) > **Other** (3) to see the Xilinx-specific options.

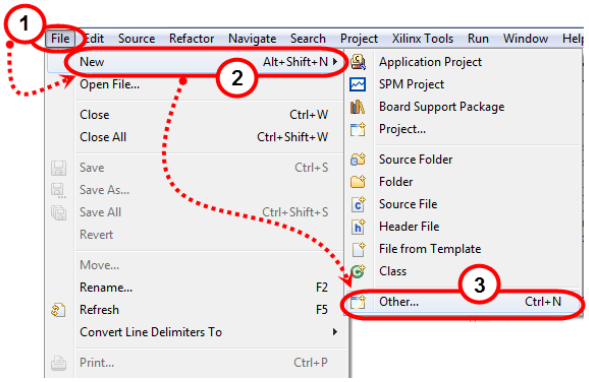


Figure 12-13: Accessing the New Wizards

The Select a Wizard dialog box opens. Here you can select one of many different wizards.

2-2-2. Expand the **Xilinx** folder (1).

2-2-3. Select **Hardware Platform Specification** (2).

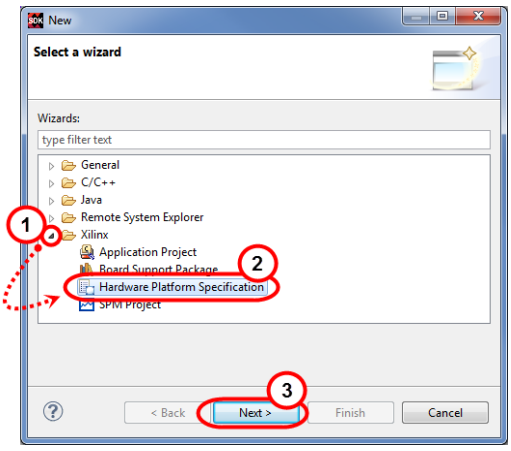


Figure 12-14: Selecting the Hardware Platform Specification Wizard

2-2-4. Click **Next** to open the New Hardware Project dialog box (3).

The New Hardware Project dialog box opens. Here you will be able to specify a project name and the hardware description file that was exported by the Vivado Design Suite.

2-2-5. Enter **UEDfull_hw** in the *Project name* field.

2-2-6. Browse to the *C:\training\deviceDriversCustom\support* directory under the Target Hardware Specification region and select the **UEDfull_[zc702 | zed].hdf** file.

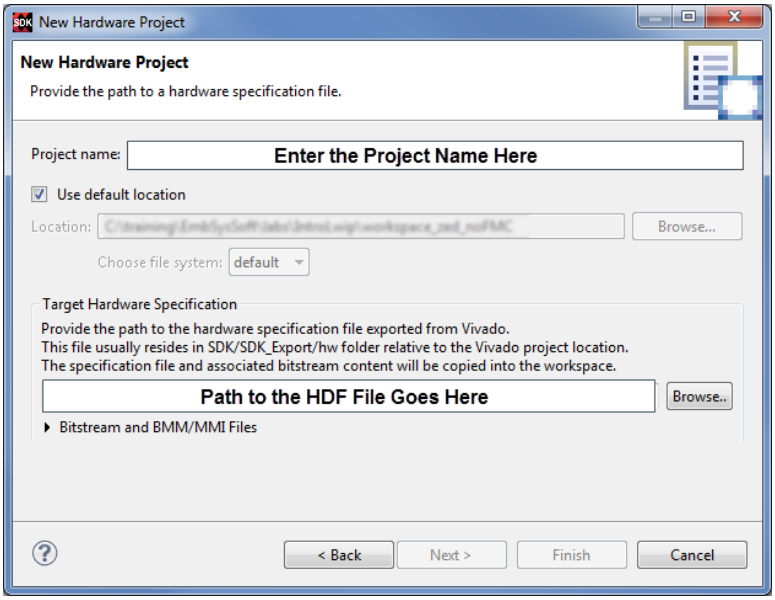


Figure 12-15: New Hardware Project Dialog Box

2-2-7. Click **Finish** to create the new hardware project.

2-3. Open the *lcd.mdd* file.

2-3-1. Select **File** > **Open File**.

2-3-2. Browse to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\data* directory.

2-3-3. Select the **lcd.mdd** file.

2-3-4. Click **Open** to open the file in a text editor.

Note: This may open in the SDK editor or whatever editor was selected as a default for opening this type of file.

2-4. Indicate that the LCD driver is going to use an instance of an axi_gpio.

2-4-1. Change *lcd* to *axi_gpio* on the line to set the OPTION to supported_peripherals to direct the association on the device driver to the name of the supported hardware.
In this case the LCD hardware is a GPIO device. This means that the drivers that are developed will give the user the impression that they are working with special LCD hardware, whereas in reality they are driving a GPIO.

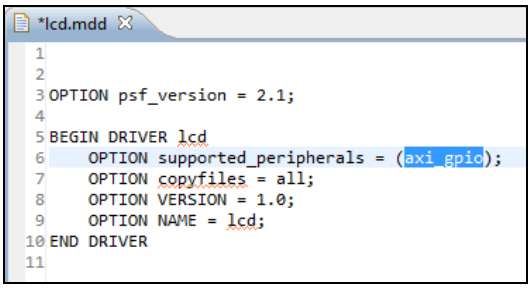


Figure 12-16: Associating the lcd Driver with the axi_gpio Peripheral

By changing this symbol, you are indicating that this driver is for the *axi_gpio* peripheral, as opposed to the non-existent LCD hardware that was specified, but not used, when the skeleton was created with the Create and Package IP Wizard.

2-4-2. Select **File** > **Save**.

2-4-3. Close the **lcd.mdd** file.

2-5. Change the parameter name in the Tcl file so that the user will have clear indications of where this peripheral can sit in memory.

2-5-1. Select **File** > **Open File**.

2-5-2. Browse to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\data* directory.

2-5-3. Open the **lcd.tcl** file.

2-5-4. Change the parameter name C_S00_AXI_BASEADDR to **C_BASEADDR**.

2-5-5. Similarly, change the parameter name C_S00_AXI_HIGHADDR to **C_HIGHADDR**.

2-5-6. Select **File** > **Save**.

2-5-7. Close the **lcd.tcl** file.
If the Tcl and MDD files were opened in an editor other than the SDK editor, you can close that editor at this time.
The parameters generated through Xilinx IPs use the naming convention of C_BASEADDR whereas the Create and Package IP Wizard uses the naming convention of C_S00_AXI_BASEADDR, which supports multiple interfaces.

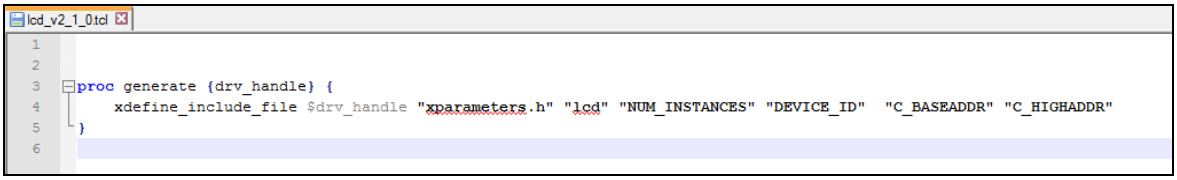


Figure 12-17: Changing the Parameter Name

2-6. Delete the *lcd.c* template file because you will be using the file already used from the "Application Development" lab.

2-6-1. Browse to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory using Windows Explorer.

2-6-2. Select the **lcd.c** file and press <**Delete**> to delete the file.

2-6-3. Click **Yes** to confirm deletion.

2-6-4. Browse to the *C:\training\deviceDriversCustom\support* directory.

2-6-5. Copy the **lcd_zynq.c** and **lcd_driver.h** files from this directory to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory.

This replaces the skeleton custom driver source file generated by the Create and Package IP Wizard with the new custom *lcd* driver source files.

Note that the file name is not important as the make utility will compile all the **.c* files in the *src* directory.

2-7. The user-defined software repositories indicate search directories for LibGen to reference when building the BSP. The location of the new driver must be included in the search path.

Note that the added path is two directories above *C:\training\deviceDriversCustom\lab*, where the custom *lcd* driver files are located. This is simply how the tools search for custom drivers.

If the wrong path is specified, the custom driver files will not be found and errors will result in the building of all software platforms and software applications that require the driver files.

Indicate the driver path for LibGen to reference when building the BSP.

2-7-1. Return to the SDK tool.

2-7-2. Select **Xilinx Tools** > **Repositories**.

2-7-3. Click **New** to the right of the Local Repositories area.

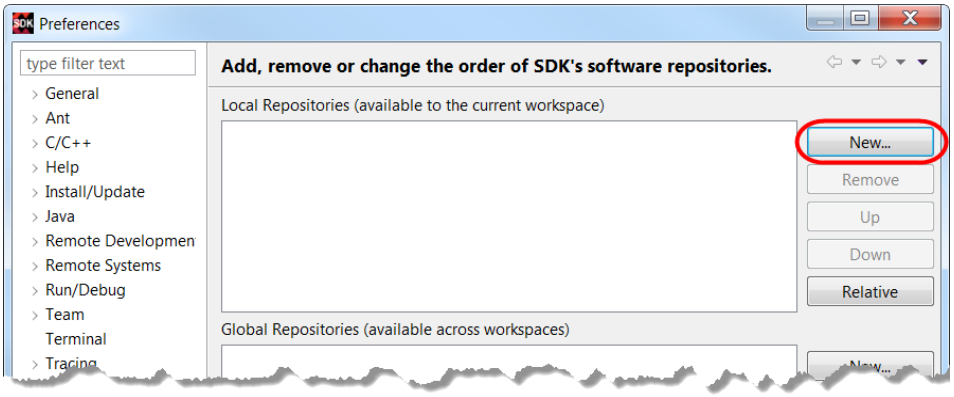


Figure 12-18: Creating a New Repository

2-7-4. Browse to and select the **C:\training\deviceDriversCustom\lab\lcd_1.0** directory.

2-7-5. Click **OK**.

The path should appear in the Local Repositories list.

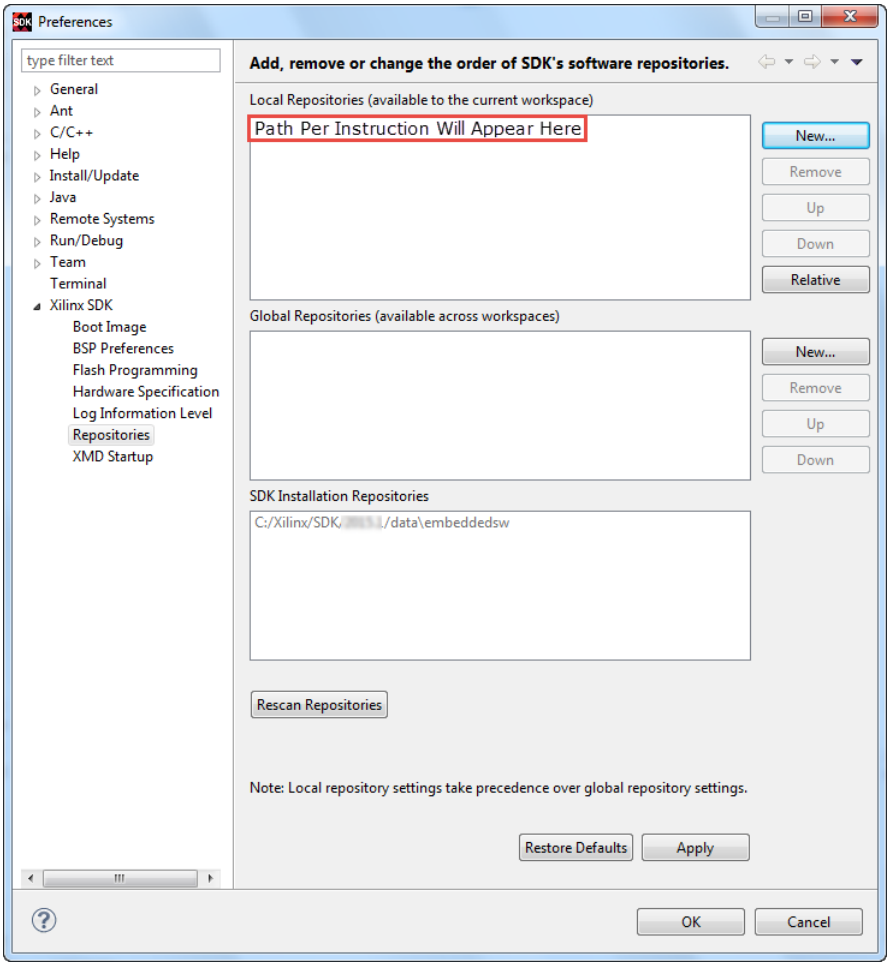


Figure 12-19: Software Repositories Dialog Box

Note: The added path is two directories above where the custom *lcd* driver files are located.

Everything in the SDK project is rebuilt because the software repository applies to the entire SDK project, not just a single BSP or software application.

2-7-6. Click **OK** to complete the software repository task.

2-8. The newly created device driver must be associated with the instance of the LCD display. Currently, the GPIO driver is selected.

You will create a new BSP named *customDriver_bsp* and set the driver for its LCD peripheral to be the new *lcd* driver.

2-8-1. Select **File** > **New** > **Board Support Package** to begin creating the BSP.

2-8-2. Enter the following settings when the Xilinx Board Support Package Project dialog box opens:

- Project name: **customDriver_bsp**
- CPU: **ps7_cortexa9_0**
- Hardware Platform: **UEDfull_hw**
- Board Support Package OS: **standalone**

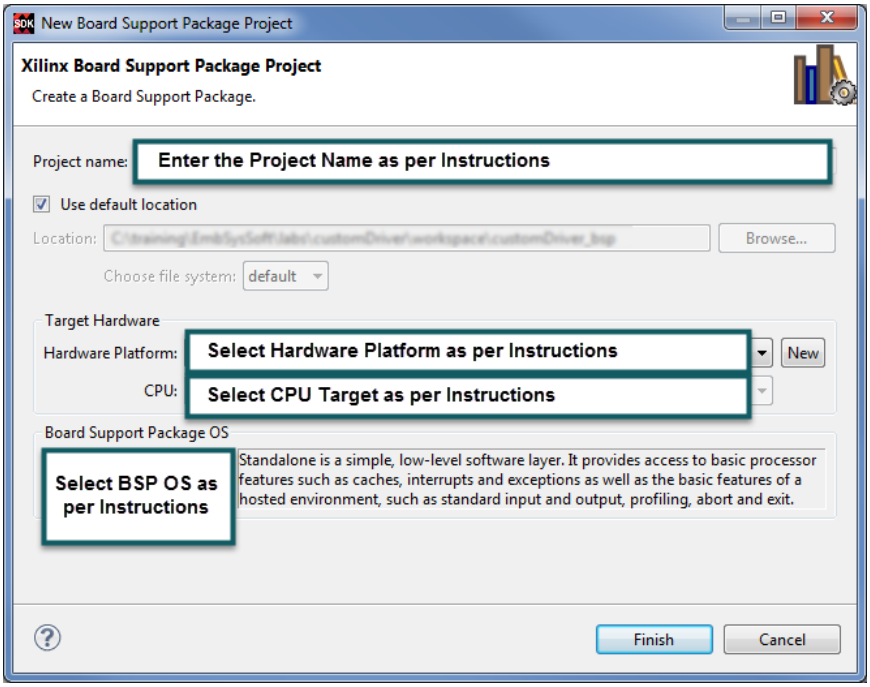


Figure 12-20: Creating a Board Support Package

2-8-3. Click **Finish**.

The Board Support Package Settings dialog box opens to enable you to customize the BSP.

2-9. Select the custom driver for the lcd peripheral.

2-9-1. Expand the **Overview** entry in the left navigation pane if it is not already expanded.

2-9-2. Select **drivers** to assign the driver for the LCD GPIO device.

2-9-3. Click the current driver for the GPIO_LCD_data component to access the down arrow for the drop-down list.

2-9-4. Click the down arrow to open the drop-down list.

2-9-5. Select **lcd** from the Driver column drop-down list.

2-9-6. Ensure that the other axi_gpio devices are designated **gpio**.

No other changes to other IP drivers is required.

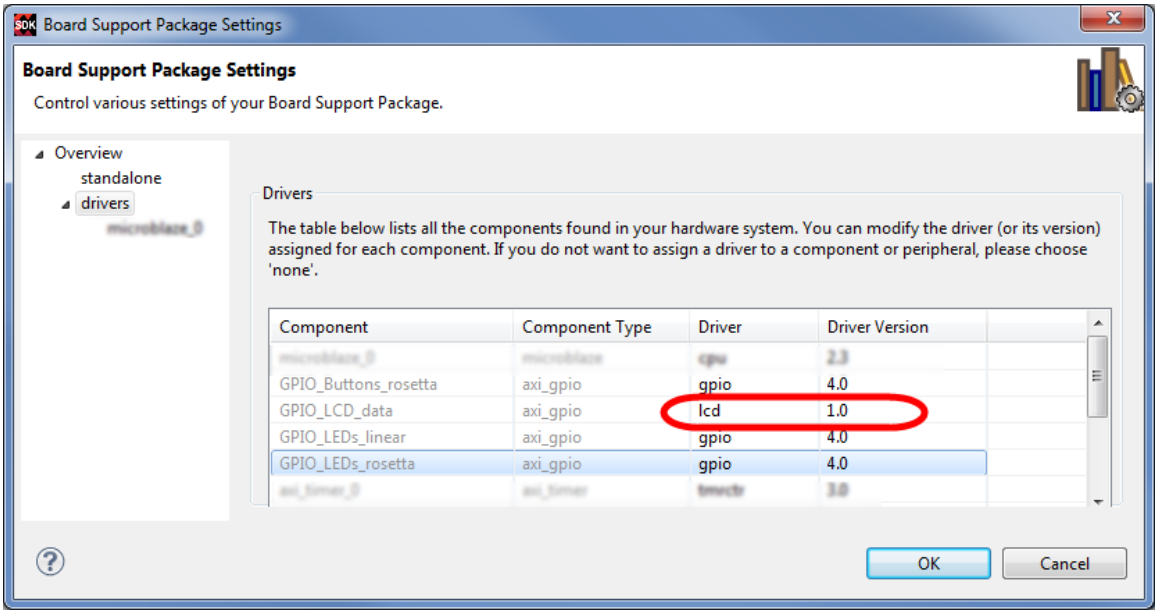


Figure 12-21: Associating the lcd Driver with the lcd_data Instance

This assigns the custom *lcd* driver that was just created to the *GPIO_lcd_data* instance in the design.

2-10. Set the compiler option to the -O0 option to reduce the optimization.

2-10-1. Expand **Overview** > **drivers** > **ps7_cortexa9_0** in the left navigation pane.

2-10-2. Enter **-O0** in the value field for *extra_compiler_flags*.

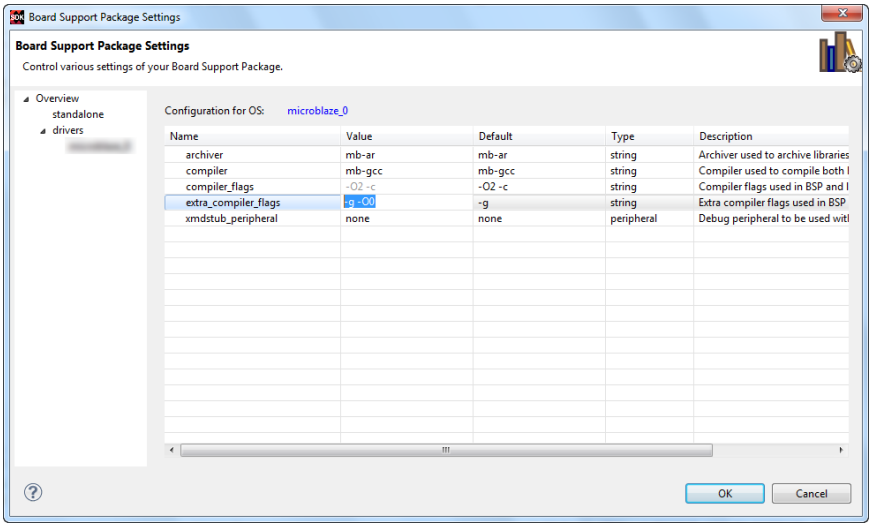


Figure 12-22: Setting the Compiler Option to -O0

2-10-3. Click **OK** to close the Board Support Package Settings dialog box and start BSP regeneration.

2-10-4. Access the Console window and verify that the BSP is built without errors.

2-11. Verify the *lcd_v1_0* directory.

2-11-1. Expand the **customDriver_bsp** project in the Project Explorer tab.

2-11-2. Verify that the *lcd_v1_0* directory was added to the processor's *libsrc* directory.

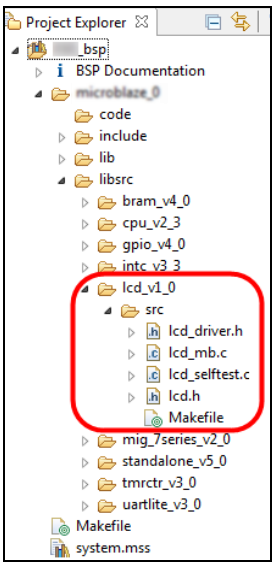


Figure 12-23: Added lcd Driver Source Files (MicroBlaze Processor)

Question 3

Which of the preceding steps resulted in *lcd* as an option for the *axi_gpio* driver in the Software Platform Settings dialog box?

Question 4

Using the Project Explorer pane, double-click the **lcd_driver.h** file to open it. Note that functions are divided into External and Internal. External functions are those called by other functions outside of the driver. What parameter is passed into all external functions? Why would this parameter be needed?

Question 5

If errors occurred in the BSP build after you added the new driver, what might the problem be?

Creating an SDK Application Step 3

The newly added driver is ready to be tested. You will use the application from the "Application Development" lab to test the driver. Note that the LCD display driver, which was part of the "Application Development" lab application project, will not be present. The same *lcd* driver is now part of the BSP. Here you will create an SDK software application project and import the *lcdDriver* source file into it.

3-1. You will create a software application project that uses the LCD by relying on the presence of the LCD drivers in the BSP.

3-1-1. Select **File** > **New** > **Application Project**.

3-1-2. Enter these settings when the New Application dialog box opens:

- Project name: **lcdDriverTest_app**
- Hardware Platform: **UEDfull_hw**
- Processor: **ps7_cortexa9_0**
- OS Platform: **standalone**
- Language: **C**
- Board Support Package: select **Use existing** and **customDriver_bsp**

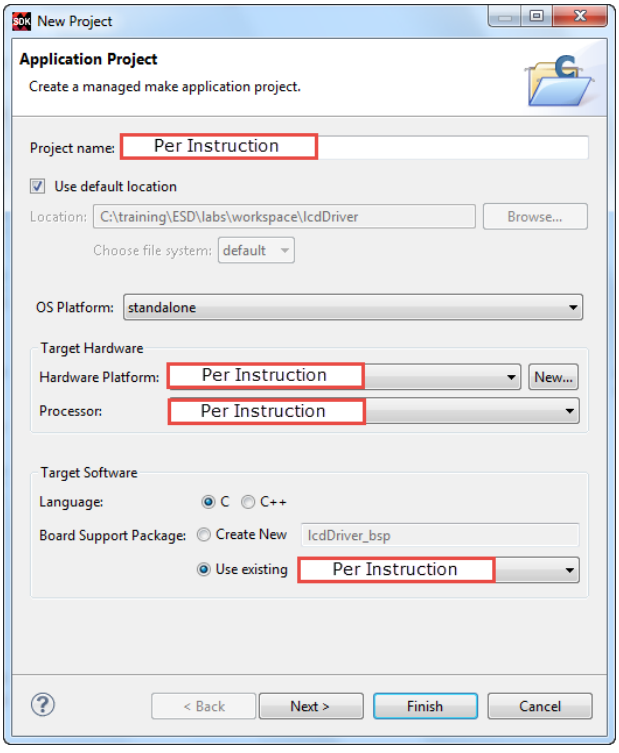


Figure 12-24: Naming the C Project and Selecting a Board Support Package

3-1-3. Click **Next**.

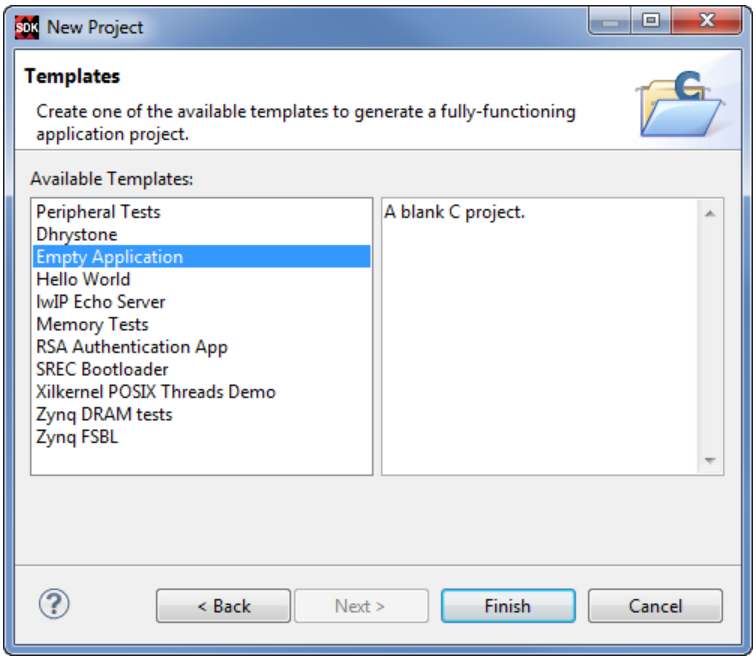


Figure 12-25: Selecting a Project Template

3-1-4. Select the **Empty Application** project template as you will be adding source files in the next set of instructions.

3-1-5. Click **Finish**.

It is common practice to add existing resource files (*.c, *.h, *.cpp, etc.) to a software project. The Eclipse framework requires this operation to be performed as an import function.

3-2. Add *stopwatch_custom_driver.c* to the application.

The preferred method for importing sources is shown here.

3-2-1. Expand the project named **lcdDriver_app** > **src** using the Project Explorer.

3-2-2. Right-click the desired destination directory in the project that you want to place the resource files (typically the src directory) (1).

3-2-3. Select **Import** to open the Import Wizard (2).

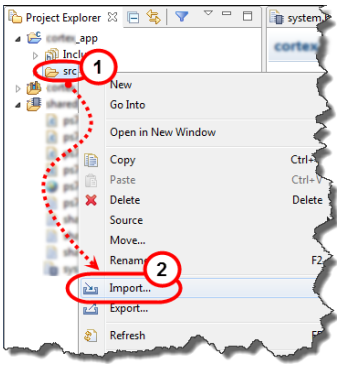


Figure 12-26: Importing a Resource File

The Import Wizard dialog box opens.

3-2-4. Expand **General** (1).

3-2-5. Select **File System** as you will be selecting individual files directly from the file system (2).

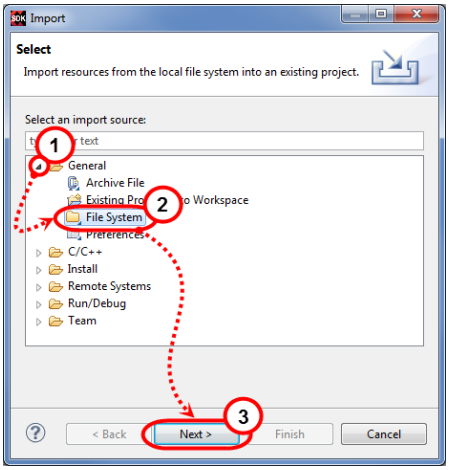


Figure 12-27: Selecting File System

3-2-6. Click **Next** to advance to specifying the files to import (3).

3-2-7. Browse to *C:\training\deviceDriversCustom\support* in the *From directory* field.

3-2-8. Select the file(s) by checking the box beside **stopwatch_custom_driver.c**.

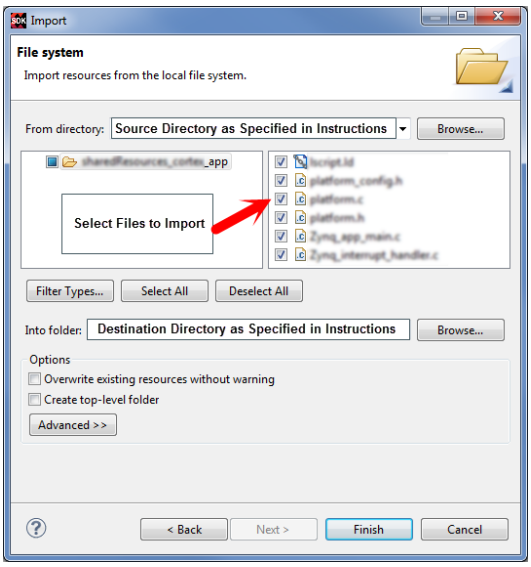


Figure 12-28: Selecting Resource Files

The *Into folder* directory will default to the location selected when you engaged the import function, but you can click **Browse** to change this location.

3-2-9. Click **Finish** to import the selected files and close the wizard.

Note: If the workspace has the automatic build option enabled, the project will automatically build with the new resource files. The Console view at the bottom of the IDE will show the results of the build.

Question 6

How can you determine that the LCD drivers are part of the BSP?

Question 7

Where can you find information about the API for this new driver set?

Open the source files for this device.

Hint: The source files are NOT under the application project. Rather, they can be found in the BSP project under the *processor name* > *libsrc* > *lcd_v1_0* > *src*.

3-3. Open *lcd_driver.h and lcd_zynq.c* in the editor.

3-3-1. Locate **lcd_driver.h and lcd_zynq.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

3-3-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

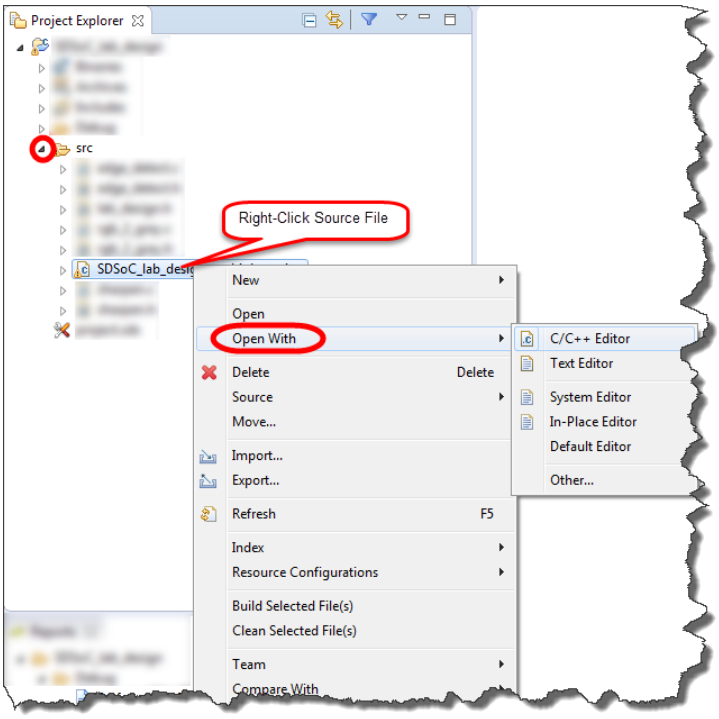


Figure 12-29: Opening a Source File via Right-Click

Question 8

What are the user callable functions?

Question 9

Where can you see the use of the API calls in *stopwatch_custom_driver.c*?

Summary

You created a new software driver for the *axi_gpio* peripheral. You then used the Create and Package IP Wizard to create the skeleton framework and used the *lcd* driver from the "Application Development" lab with minor modifications as the main source for this driver.

Answers

1. Browse to the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory. This directory is where the source files for the device driver reside. What files have been placed there by the wizard?
   - *lcd.c*
   - *lcd.h*
   - *lcd_selftest.c*
   - *Makefile*
2. What is in the *lcd.c* file?

   Nothing. It is a shell.
3. Which of the preceding steps resulted in *lcd* as an option for the *axi_gpio* driver in the Software Platform Settings dialog box?

   Changing the supported peripherals option in the *lcd.mdd* file from *lcd* to *axi_gpio*.
4. Using the Project Explorer pane, double-click the **lcd_driver.h** file to open it. Note that functions are divided into External and Internal. External functions are those called by other functions outside of the driver. What parameter is passed into all external functions? Why would this parameter be needed?

   The LCD base address is passed into all external functions. It is needed because the LCD base address will vary with hardware configuration. Passing the base address as a parameter is one way to get this necessary information into the driver functions.
5. If errors occurred in the BSP build after you added the new driver, what might the problem be?

   An error in one of the C source files located in the *C:\training\deviceDriversCustom\lab\lcd_1.0\drivers\lcd_v1_0\src* directory or the driver source files were not found due to an incorrect or missing path in the User-Defined Software Repositories.
6. How can you determine that the LCD drivers are part of the BSP?

   The easiest way to tell is that the LCD driver files are not found in the application source file folder but are found in the following locations:

   Zynq AP SoC users: *customDriver_bsp* > *ps7_cortexa9_0* > *libsrc* > *lcd_v1_0* > *src* folder, which is a BSP project.

   MicroBlaze processor users: *customDriver_bsp* > *microblaze_0* > *libsrc* > *lcd_v1_0* > *src* folder, which is a BSP project.

7. Where can you find information about the API for this new driver set?

   Unless you write your own API documentation, the source files are the only imported documentation.

8. What are the user callable functions?

   lcd_driver.h provides a number of macros (Level 0 drivers):*LCD_mWriteReg*, *LCD_mReadReg*, and *LCD_Reg_SelfTest*.

   lcd_zynq.c provides higher-level functions and macros, including *mLCDsetDataDir*, *mLCDgetDataReg*, and *mLCDsetDataReg*, as well as *WriteInst()*, *WriteData()*, *LCDon()*, *LCDoff()*, and *LCDcursorOn()*.

9. Where can you see the use of the API calls in *stopwatch_custom_driver.c*?

   LCDInit and *LCDcursorOff()* are called near line 148, and *LCDClear()* and *LCDPrintString()* are found near line 222.