

Software Interrupts: Writing

2021.2

Abstract

Interrupts provide a low-latency response to events. This lab demonstrates how to replace a polled approach (a software timing loop) with an interrupt-driven technique.

This lab should take approximately 60 minutes.

CloudShare users only: You are provided three attempts to access a lab, and the time allotted to complete each lab is 2X the time expected to complete the lab. Once the timer starts, you cannot pause the timer. Also, each lab attempt will reset the previous attempt—that is, your work from a previous attempt is not saved.

Objectives

After completing this lab, you will be able to:

- Navigate Xilinx processor device services documentation
- Locate the device driver API documentation
- Use the BSP processor interrupt services
- 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 easy to implement and acceptable for certain types of applications, there is a very significant disadvantage: latency.

Latency is the amount of time between when an event occurs and when the software begins responding to it. If the latency is not predictable, then the outcome or system behavior may become unpredictable. The interrupt process solves this problem.

An interrupt is a connection from a device 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 have you complete an interrupt-based stopwatch application. The application program has been mostly written for you and includes use of the general interrupt controller (GIC) and triple timer counter (TTC) peripherals for the Zynq® UltraScale+™ MPSoC and Versal® devices.

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 triple timer counter device driver services.

Note: For the 2021.2 release, it is strongly recommended that the MPSoC device be used as the QEMU emulator sometimes has issues with cleanly launching and running code targeting the Versal devices.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform.

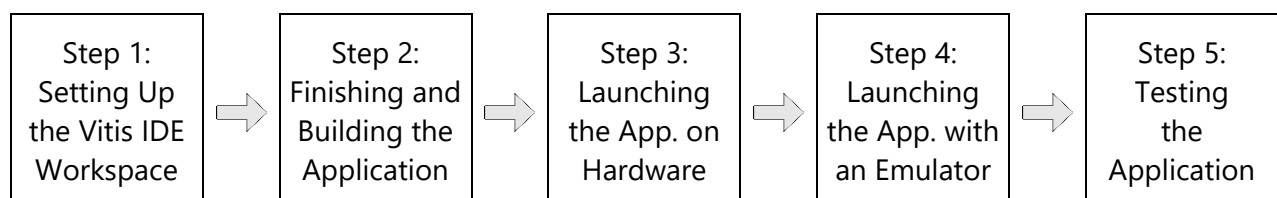
One environment variable is required: `TRAINING_PATH`, which points to where the lab files are located. This variable comes configured in the CloudShare/CustEd_VM environments.

Some tools can use this environment variable directly (that is, `$TRAINING_PATH` will be expanded), and some tools require manual expansion (`/home/xilinx/training` for the CloudShare/CustEd_VM environments). The lab instructions describe what to do for each tool.

Both the Vivado Design Suite and Vitis platform offer a Tcl environment that is used by many labs. When the tool is launched, it starts with a clean Tcl environment with none of the procs or variables remaining from a previous launch of the tools.

This means that if you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool, you will need to source the Tcl script again and set any variables that the lab requires. This is also true of terminal windows—any variable settings will be cleared when a new terminal opens.

General Flow




Setting Up the Vitis IDE Workspace

Step 1

You will begin this lab by opening the Vitis IDE and running a Tcl script that will set up the starting point for stopwatch application.

1-1. Launch the Vitis IDE and set the workspace.

1-1-1. Click the **Vitis IDE** () icon from the taskbar to launch the tool.

Hint: There are different versions of the Vitis tools for different development purposes. Make certain that you select the Vitis IDE represented by the icon shown. The Vitis HLS tool has a similar icon but with a gear in the lower right corner. Many tasks CANNOT be run with the Vitis HLS tool—only from the Vitis IDE, so make certain that you select the correct icon.

The Workspace Launcher opens.

The Vitis IDE creates a workspace that initially only contains a thin structure that tracks tool settings and maintains the tool log file. As projects are added, this workspace will update to include all types of projects. Workspaces can be switched from within the tool by selecting **File > Switch Workspace**.

Note: The Vitis IDE can be launched from a Linux terminal window and requires an additional step of sourcing a script to configure the environment.

Open a Linux terminal by pressing **<Ctrl + Alt + T>** or clicking the Terminal icon in the toolbar and entering the Tcl command **source <space> <path to the Vitis IDE tool>/settings64.sh** to set up the environment. Then, once this setup script completes, the tool can be launched by entering **<path to the Vitis IDE tool>/bin/vitis**.

Note: The customer training environment (CustEd_VM) installs the Vitis tool to `/opt/Xilinx/Vitis/2021.2`. If the Vitis tool is installed in a different location in your environment, use that tool path.

Similarly, the Vitis IDE tool can be launched in Windows by opening a command shell and entering **<path to the Vitis IDE tool>/settings64.bat**, then **<path to the Vitis IDE tool>/bin/vitis**. For obvious reasons, most users choose to launch the tool from the taskbar or desktop icon.

1-1-2. Enter **\$TRAINING_PATH/SWinterrupts/lab** into the Workspace field or use the Browse button to browse to the location (1).

Note: This tool does not support the expansion of environment variables, so you will perform this expansion yourself. Replace `$TRAINING_PATH` with the path to the training directory followed by `SWinterrupts/lab`.

If you do not know what the path to the training directory is, you can easily find out by:

[Linux]: Opening a terminal (**<Ctrl + Alt + T>**) and entering **echo \$TRAINING_PATH**.

[Windows]: Opening a command prompt shell (press <**Windows key** + **R**> and enter **cmd** in the Open field) and entering **echo %TRAINING_PATH%**.

The default path for the provided Customer Training VM is `/home/xilinx/training`; however, your machine may have been configured differently.

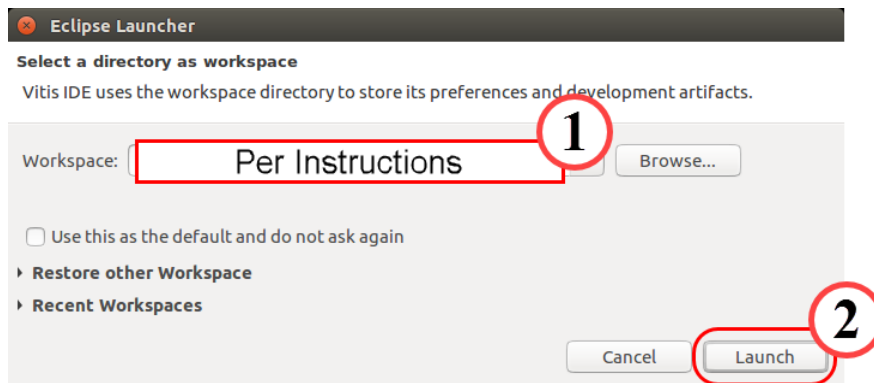


Figure 8-1: Selecting the Workspace

1-1-3. Click **Launch** to and open the new workspace (2).

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

Ensure that the tool window is maximized so that you can see all the view panes.

The XSCT console is available in the Vitis IDE as a command terminal window. This console enables you to enter Tcl commands directly into the environment. Commands can be entered as scripts or as individual commands. The XSCT tool is available as a command-line tool and enables you to configure Vitis tool projects without the use of the GUI.

Having Tcl commands available is especially useful when you are building a design up to a particular point, performing repetitive processes, or rebuilding a design from a well-documented, step-by-step process.

Here you will explore using the XSCT console in the Vitis IDE tool.

- 1-2. Open the XSCT console so that you can enter Tcl-based commands directly into the Xilinx software development tool.

The process shown here illustrates how any view can be opened, using the XSCT Console as an example. If you are already familiar with this process, you can click the XSCT Console icon (🖥️) to bypass the lengthier process and continue with the next instruction.

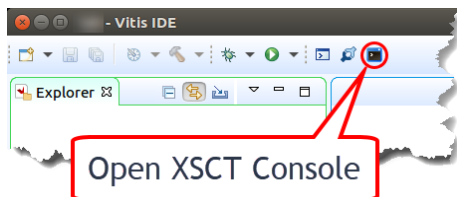


Figure 8-2: Opening the XSCT Console

- 1-2-1. Select **Window** (1) > **Show view** (2) to see a list of commonly opened views.

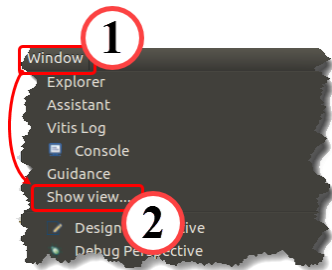


Figure 8-3: Selecting Show View

- 1-2-2. Expand the **Xilinx** entry to see the Xilinx-specific views (3).

- 1-2-3. Select **XSCT Console** (4).

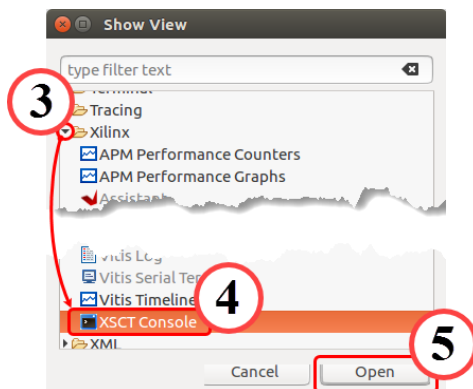


Figure 8-4: Locating the XSCT Console

- 1-2-4. Click **Open** to open the XSCT console (5).

Typically, this console opens next to the group of tabs in the Build console. You can also click-drag the tab to other locations or resize it for easier use.

You will now run a Tcl script that will set up your workspace and projects. This script creates the platform project, a domain for one of the APU processors, and a system project that contains an application project.

1-3. Run the provided Tcl script.

Enter the following Tcl commands into the Tcl command line when using the XCST console.

- 1-3-1.** If the XSCT console is not open in the Vitis IDE, click the XSCT icon () in the toolbar.

The XSCT console can also be opened via **Xilinx > XSCT Console**.

- 1-3-2.** Enter the following command to change the directory to the location of the Tcl script you want to run:

```
cd $::env(TRAINING_PATH)/SWinterrupts/support
```

The `$::env()` is needed as `SWinterrupts` is defined as an environment variable. This is Tcl's way of importing a value from the environment into the Tcl shell.

- 1-3-3.** Enter the following command to run the script:

```
source SWinterrupts_builder.tcl
```

This will load the environment with the Tcl *procs* contained in `SWinterrupts_builder.tcl` and run any Tcl code not included in the procs.

- 1-3-4.** Enter the following into the Tcl command line to select the platform:

```
[MPSoC users]: boardSelect ZCU104
```

```
[Versal ACAP users]: boardSelect VCK190
```

- 1-3-5.** Build the starting point for this lab based on the board selected:

```
buildStartingPoint
```

This takes several minutes to run.

- 1-3-6.** If you are planning to use the QEMU emulator, enter the following:

```
useQEMU
```

This sets a compiler symbol that slightly modifies the code to run better with QEMU.

1-4. Confirm the proper build settings.

1-4-1. Click the arrow (▼) next to the Hammer/Build icon (🔨).

Hint: If the Hammer icon is grayed out, select a project.

1-4-2. Select **Debug**.

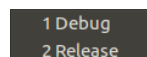


Figure 8-5: Build Choices

This sets the default configuration for all future builds. This value can be changed at any time.

When you select a configuration, the compiler is launched.

Note: You can build the project directly by right-clicking the project (1) and selecting **Build Project** from the context menu (2).

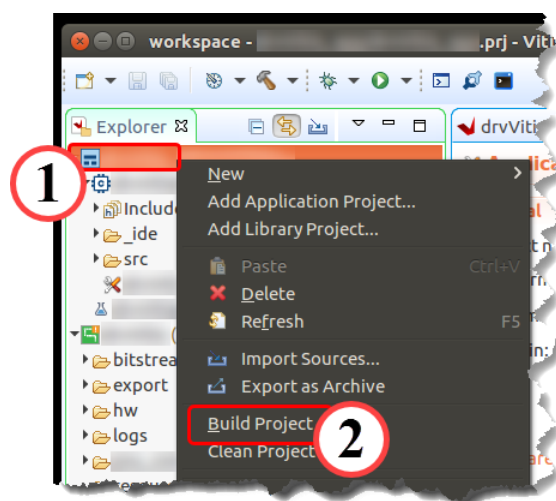


Figure 8-6: Building Projects

Alternatively, you can click the **Hammer/Build** icon (🔨).

The important part about this instruction is that it will build all the platform support code. The following step will have you add two calls to enable the interrupts, and then you will rebuild the project there.

Since the platform support code contains the BSPs and other support code, it will take a few minutes to run. When you modify the source code, building will be very quick.

Advance to the next step and continue as you wait for this longer build to complete.

Finishing and Building the Software Application

Step 2

The stopwatch application has been provided to you nearly completed. What remains is the enabling of the timer's interrupt and making the interrupt controller and processor sensitive to this interrupt. These actions will need to be taken regardless of the system that the interrupt is being implemented in.

There are several 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.

You will begin 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.

For the PS-based devices there is the GIC, which takes interrupts from a wide number of sources, including the peripherals in the IOP block, the PL, and SCU. Then there is the vector interrupt table in this device that maps a handler to each interrupt source. The other place sensitive to interrupts is the CPU itself. The CPU has relatively few interrupt inputs (IRQ and FIQ); however, it maintains its own vector table which includes the entry for interrupts.

While this lab focuses on devices containing a processing system block, a quick comparison with the MicroBlaze processor is not out of line. The MicroBlaze processor behaves similarly, but instead of having a GIC, it has an interrupt controller peripheral. The MicroBlaze processor contains its own vector table and, for all practical purposes, a single interrupt input.

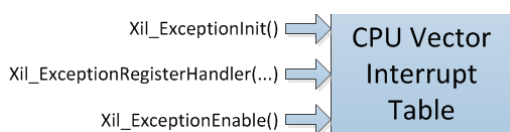


Figure 8-7: Interrupt Targets and Some Associated Function Calls (MPSoc PS/GIC)

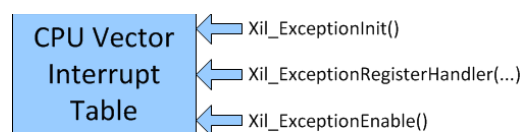
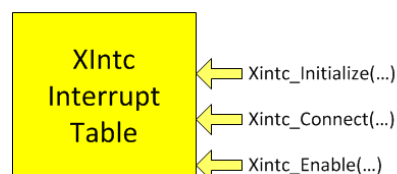
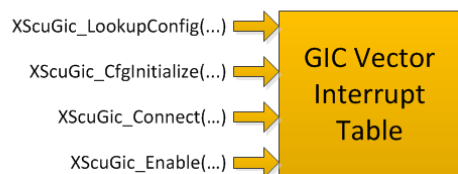


Figure 8-8: Interrupt Targets and Some Associated Function Calls (MicroBlaze Processor and INTC)



The functions are aptly named in that the ones associated with the GIC (used with devices containing a PS) begin with "XScuGic_..." (Xilinx snoop control unit - general interrupt controller); those associated with the interrupt controller peripheral (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 GIC calls must be made in the following 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.

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 Ultrascale+ MPSoC and Versal devices, the next block of code manages the setup for the triple timer counter (TTC), which does not directly apply to how interrupts are implemented in the processor; rather it is the interrupt source in this design.

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.

- XTtCps_EnableInterrupts – Enable the timer's interrupt to the interrupt controller.
- XScuGic_Enable – 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. Expand **SWIntr_plat** in the Explorer window.

2-1-2. Double-click **platform.spr** to open the platform project (1).

2-1-3. Expand per your platform (2):

[MPSoC users]: SWIntr_plat > psu_cortexa53_0 > SWIntr_dom

[Versal ACAP users]: SWIntr_plat > versal_cips_0_pspmc_0_psv_cortexa72_0 > SWIntr_dom

2-1-4. Select **Board Support Package** under SWIntr_dom (3).

The Board Support Package window opens.

2-1-5. Select the **Drivers** tab in the Board Support Package window (4).

Note: You may need to enlarge or maximize the swIntr_plat tab to find the peripherals in the next two tasks.

2-1-6. Click the **Documentation** link adjacent to the *psu_acpu_gic* [MPSoC] or *versal_cips_0_pspmc_0_psv_acpu_gic* [Versal ACAP] peripheral (5).

2-1-7. Click the **Documentation** link adjacent to the `psu_ttc_0` [MPSoC] or `versal_cips_0_pspmc_0_psv_ttc_0` [Versal ACAP] peripheral (6).

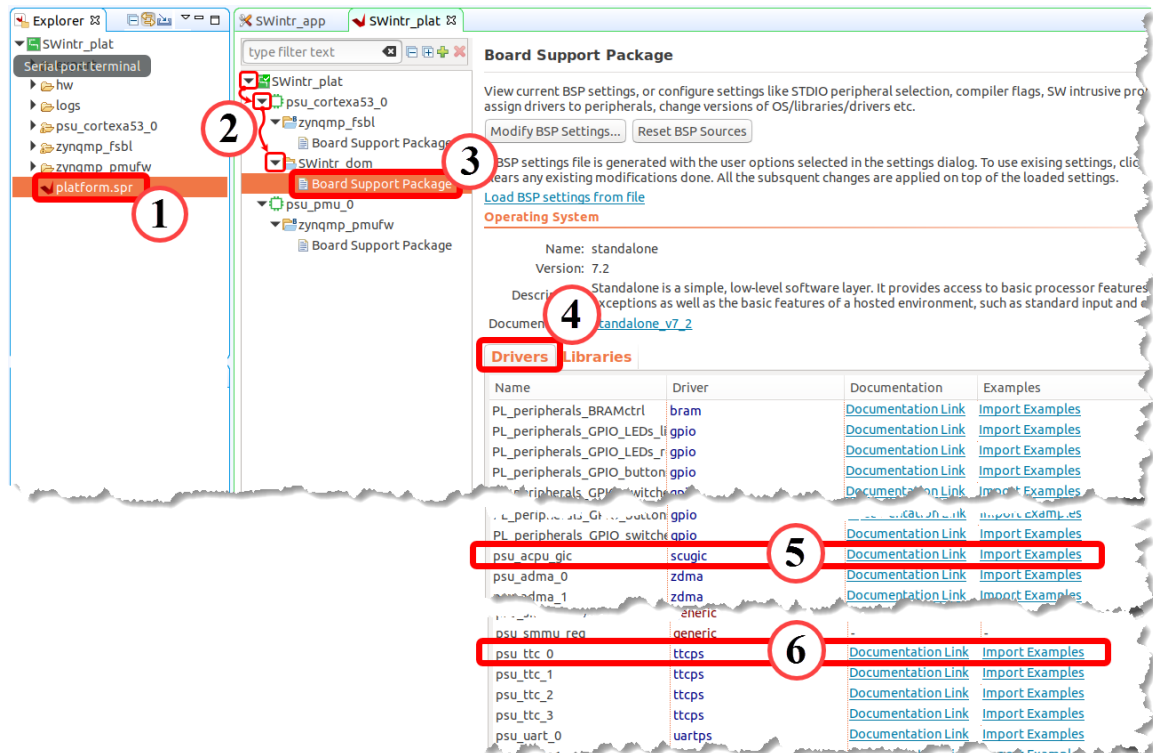


Figure 8-9: Accessing the Driver Documentation

A browser window opens to the Overview tab for the peripheral.

2-2. View the *scugic* 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**.

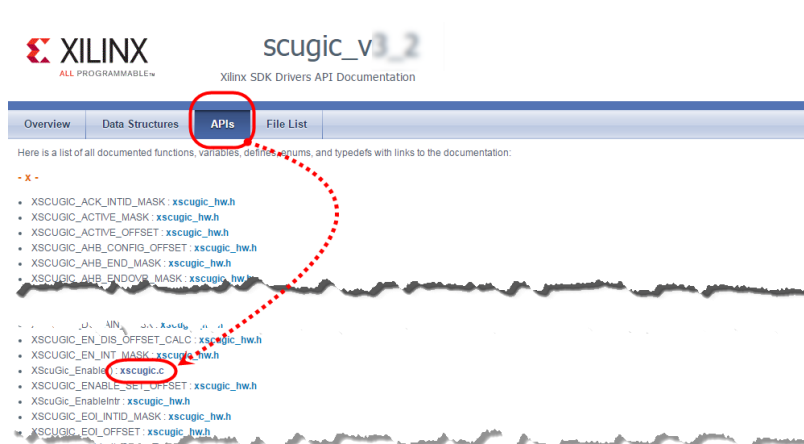


Figure 8-10: Locating the API for the SCUGIC

2-2-3. Click the **xscugic.c** link and view the function details.

```
void XScuGic_Enable(XScuGic *InstancePtr,
                  u32      Int_Id
                  )
    Enables the interrupt source provided as the argument Int_Id. Any pending interrupt condition for the specified Int_Id will occur after this function is called.
```

Parameters:
InstancePtr is a pointer to the *XScuGic* instance.
Int_Id contains the ID of the interrupt source and should be in the range of 0 to XSCUGIC_MAX_NUM_INTR_INPUTS - 1

Returns:
 None.

Note: None.

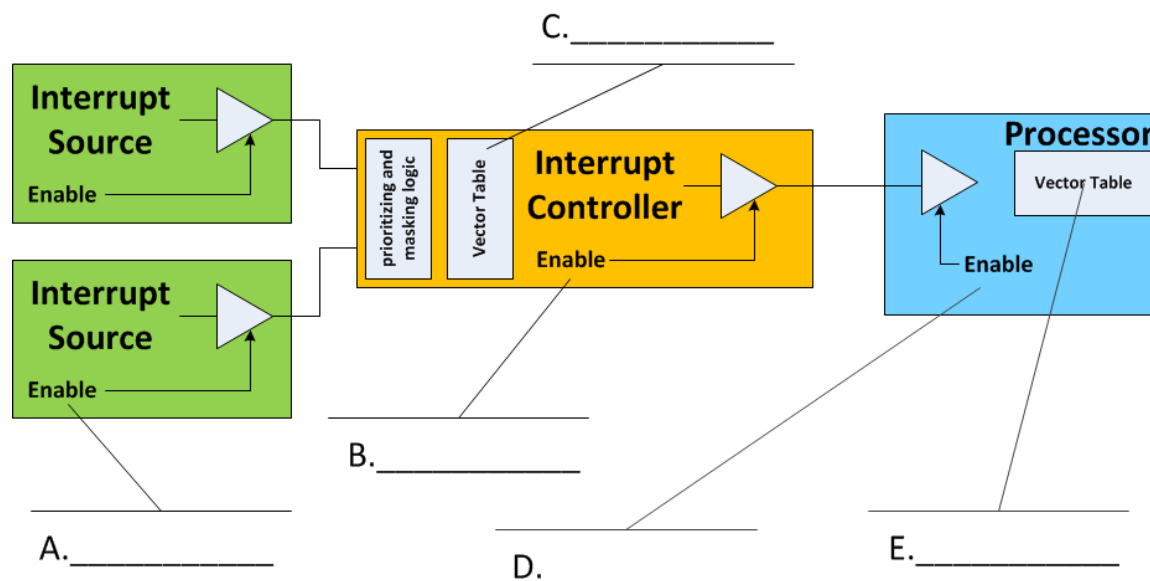
Figure 8-11: Viewing the XScuGic_Enable Function

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 8-12: 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 **SWintr_sys** > **SWintr_app** > **src** to access the list of sources in the application project.

This can be done by clicking the small triangle next to the name (🔍). You will need to click this icon next to SWinterrupt_app, then again next to src.

- 2-3-2.** Double-click **stopwatch_intr.c** to open it in the editor.

- 2-3-3.** Locate the comment "*manage the SCU's GIC*" on or near line 63.

If you do not recall how to perform a search, refer to the *Lab Reference Guide* > Vitis IDE Operations > Finding Text in a Source File.

Here you will see that the general interrupt controller (GIC) is initialized.

- 2-3-4.** Locate the comment "*Manage the timer*" on or near line 89.

Here you will see that the Triple Timer Counter (TTC) is initialized.

- 2-3-5.** Review the code and answer the questions.

Note: There are two rough flow diagrams (PNG files) provided in the `$TRAINING_PATH/SWinterrupts/support` directory for better understanding of the code.

Question 2

What two important operations are missing from this code?

- 2-3-6.** Test your skills and write the code to enable the interrupt from the GIC.

Using the documentation referenced above, make an effort to determine the proper function call. Add your code snippet after the line commented by:

```
// Enable the SCU GIC interrupt here
```

If you get stuck, there is a code snippet at the end of the header comment that solves this problem.

- 2-3-7.** 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:

```
// Enable the timer interrupt here
```

If you get stuck, there is a code snippet at the end of the header comments that solves this problem.

2-3-8. Save the **stopwatch_intr.c** file.

Because the application uses the *printf()/xil_printf()* functions, heap space must be allocated to them so that they behave properly.

2-4. Modify the linker script to use at least 4K of memory for the heap and run the application from the DDR memory.

2-4-1. Right-click the application to open the context menu.

2-4-2. Select **Generate Linker Script**.

2-4-3. Set the Code, Data, and Heap and Stack sections to the DDR memory.

Note: MPSoC users should target *psu_ddr_0_MEM_0*, and Versal ACAP users should target *axi_noc_0_CO_DDR_LOW0*.

2-4-4. Set the heap size to **4 KB**.

2-4-5. Click **Generate** to build the new linker script.

2-4-6. If you are asked if you want to overwrite the existing linker script, click **Yes**.

2-5. Confirm the proper build settings.

2-5-1. Click the arrow (▼) next to the Hammer/Build icon (🔨).

Hint: If the Hammer icon is grayed out, select a project.

2-5-2. Select **Debug**.

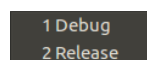


Figure 8-13: Build Choices

This sets the default configuration for all future builds. This value can be changed at any time.

When you select a configuration, the compiler is launched.

Note: You can build the project directly by right-clicking the project (1) and selecting **Build Project** from the context menu (2).

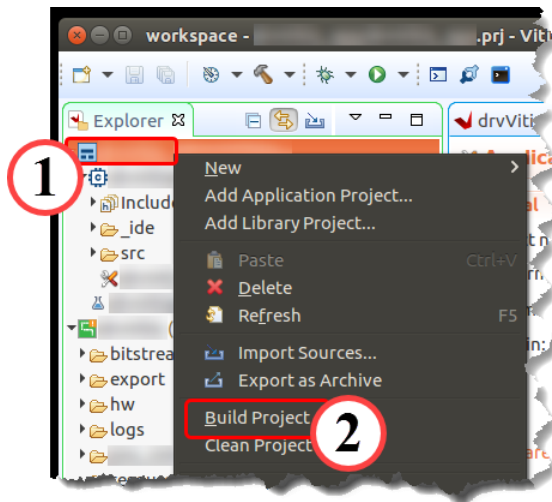



Figure 8-14: Building Projects

Alternatively, you can click the **Hammer/Build** icon ().

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?

Launching the Application on Hardware

Step 3

If you have a ZCU104 available, you can follow these instructions to use it. If you do not have a board, you can skip to the next step, "Launching the Application with an Emulator".

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. Bring up the ZCU104 board.

- 3-1-1.** Ensure that the board is powered off and that the board is properly configured and connected.

Note: This lab will have you boot using JTAG mode, which requires the mode pins to be set as described below.

- 3-1-2.** Locate SW6 on the board.

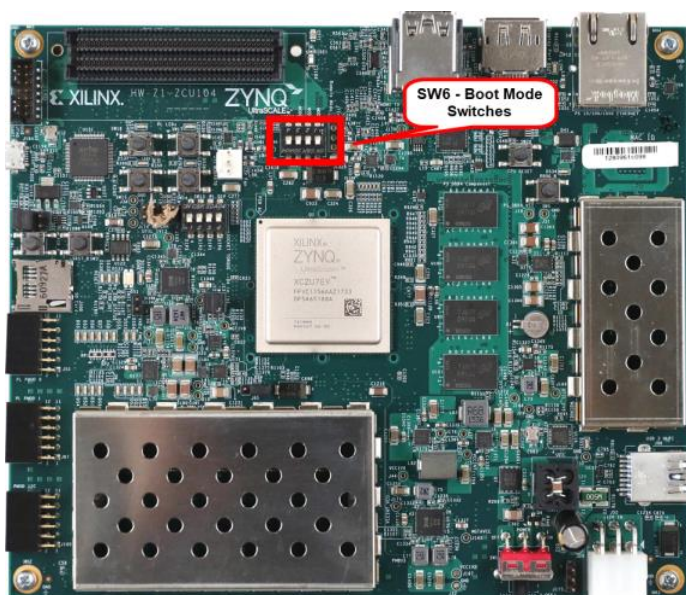


Figure 8-15: ZCU104 SW6 - Boot Mode Switches

- 3-1-3.** Set SW6 as shown below to ensure that the board is configured to boot from **JTAG**.

Note: Settings are shown to illustrate different boot mode settings. Make sure you select JTAG.

Boot Mode	Mode Pins [3:0]	Mode SW6 [4:1]
JTAG	0000/0x0	ON,ON,ON,ON
QSPI32	0010/0x2 ⁽¹⁾	ON,ON,OFF,ON
SD1	1110/0xE	OFF,OFF,OFF,ON

Figure 8-16: Board Configuration



Figure 8-17: ZCU104 Boot Settings

3-1-4. Slide the power switch to the "ON" position to power on the board.

3-2. Determine which COM port connects to the USB serial port on the development board.

3-2-1. If not already done, connect the power and USB cables to the board.

Since the serial communications occurs over a USB connection, the USB port must be enumerated in order for the serial port to be seen which requires the board to be powered on.

3-2-2. Identify the appropriate channel for the USB as follows:

For those using the Xilinx Customer Education VM, select **Devices > USB > Xilinx JTAG+3Serial** to ensure that the VM has access to the USB connection.

Note: The VirtualBox Extension Pack is required to access the USB ports from the VM. Commercial use requires purchasing a license.

Hint: This is done from the virtual machine shell, not from within the VM itself. Look for the menu at the top of the VM window.

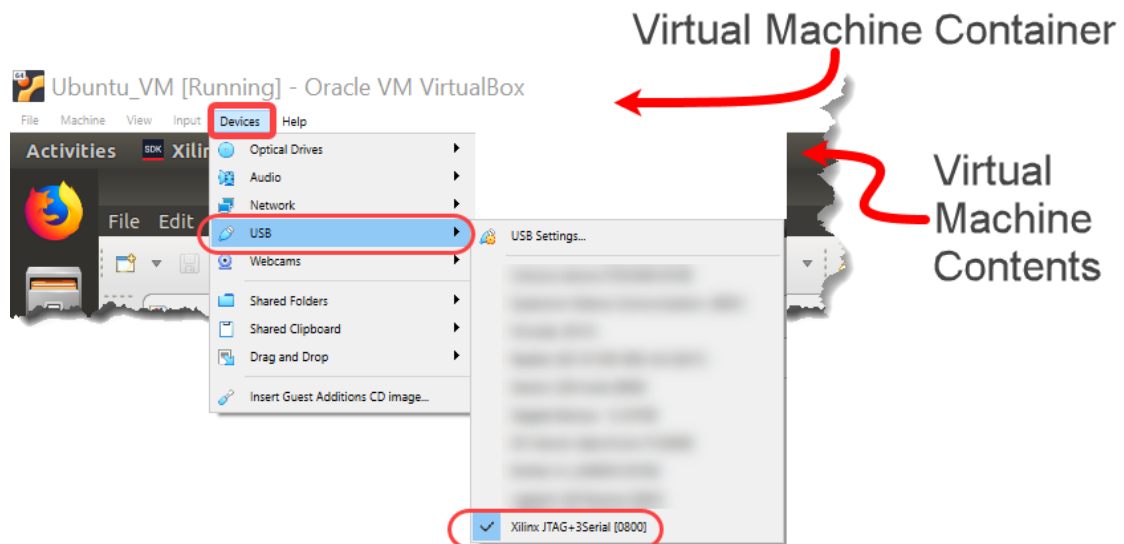


Figure 8-18: Determining the COM Port in the VirtualBox Environment

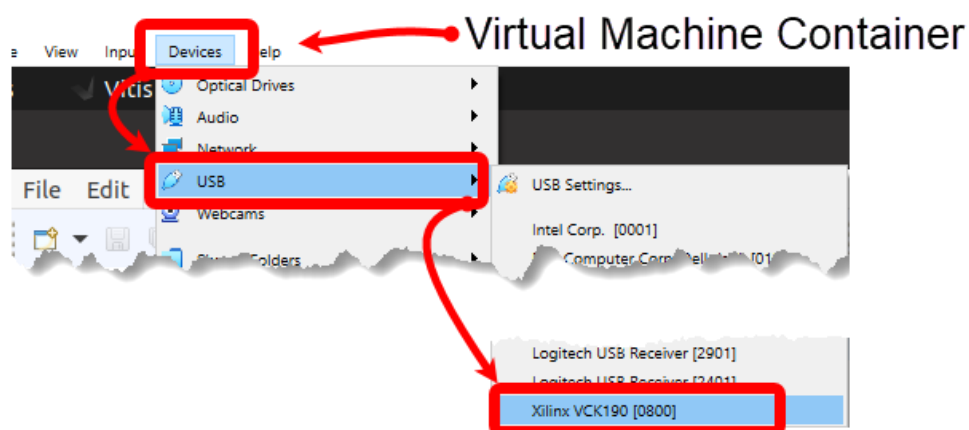


Figure 8-19: Accessing the USB Port When the VCK190 Is Connected

If you are not using the provided VM, you will need to identify the proper serial port to use based on your operating system.

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

3-3. Locate the Vitis Serial Terminal tab.

3-3-1. If the Vitis Serial Terminal tab is not currently visible, select **Window** (1) > **Show view** (2) to see a list of commonly opened views.

3-3-2. Expand the **Xilinx** entry (3).

3-3-3. Select **Vitis Serial Terminal** (4).

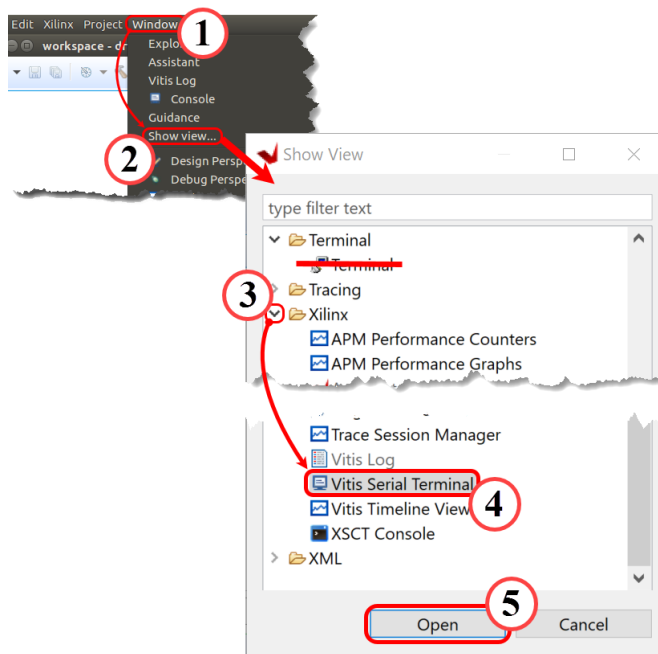


Figure 8-20: Opening the Vitis Serial Terminal

3-3-4. Click **Open** to access the Vitis serial terminal (5).

This tab typically opens adjacent to the Console tab. You can leave this terminal in this position or drag the tab to another window region.

3-4. Configure the Vitis serial terminal.

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

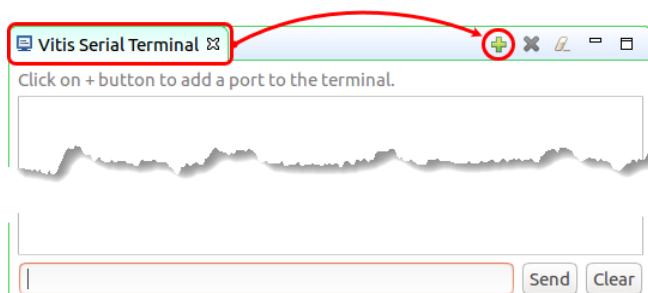


Figure 8-21: Adding or Associating a Port to the Terminal

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. Your board must be powered on to see this port.

Many Xilinx evaluation boards use a multi-ported USB serial port bridge that contain four serial ports, three COM ports, and one serial port dedicated to the JTAG chain(s). If this is the case, the three COM ports will be in numerical order.

You should select the lowest number COM port that is attached to PS uart 0.

Note: If you are using the CustEd VM, this will be `/dev/ttyUSB1`. This requires that the Extension Pack is installed (read the EULA).

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

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

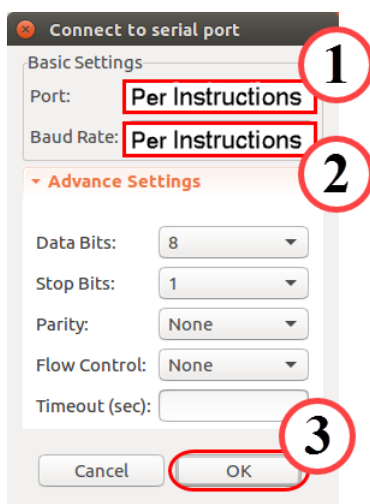


Figure 8-22: Configuring the Terminal

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

Configurations are a powerful ally in setting up a run for specific sets of criteria. The default configuration settings are adequate for most basic programs and do not require clicking through multiple dialog boxes just to run the program.

3-5. Run **SWintr_app** with the default configuration settings.

3-5-1. Right-click **SWintr_app** from the Explorer pane (1).

3-5-2. Select **Run As** from the context menu (2).

3-5-3. Select **Launch on Hardware (Single Application Debug)** to immediately launch the application on the hardware (3).

Note: Selecting **Launch on Hardware (GDB)** will also work; however, this is a deprecated flow.

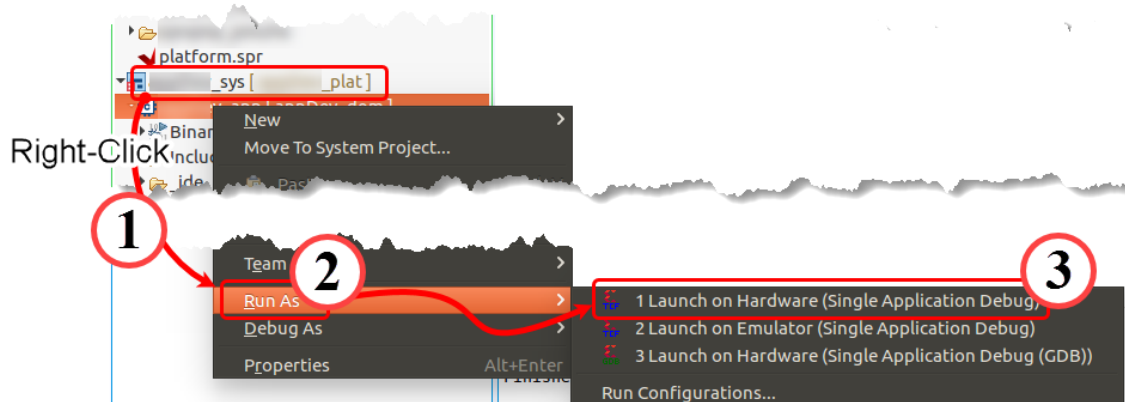


Figure 8-23: Launching the Application on Hardware Using the Default Configuration Settings

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.

--- Alternative Method ---

Select **SWintr_app** from the Explorer pane.

Once the application project is selected, click the **Run** icon (▶) and then select the type of run as explained above.

3-5-4. Skip to the final step, which is "Testing the Application".

Launching the Application with an Emulator

Step 4

Configurations are a 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 dialog boxes just to run the selected application.

4-1. Run **SWintr_app** in the emulator with the default configuration settings.

4-1-1. Right-click **SWintr_app** from the Explorer pane (1).

4-1-2. Select **Run As** from the context menu (2).

4-1-3. Select **Launch on Emulator (Single Application Debug)** to immediately launch the application on the hardware (3).

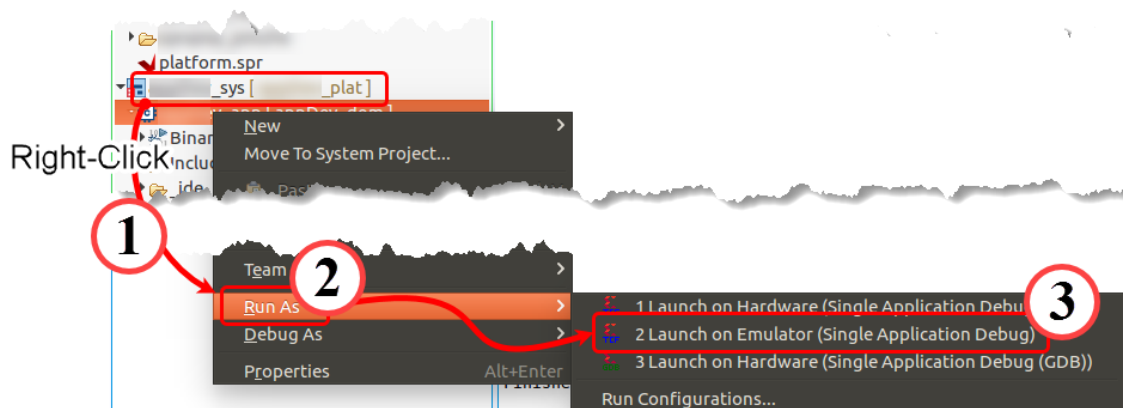


Figure 8-24: Running the Application in the Emulator with the Default Configuration

If an application is already running under emulation, a warning message will appear asking you to terminate the previous session. If you receive this message, click **Yes**, which starts the new scenario.

If the application was not running under emulation, you will receive a message requesting permission to launch the emulator and run the application.

--- Alternative Method to Launch the Emulator ---

Select **SWintr_app** from the Project Explorer pane.

Once the application project is selected, click the **Run** icon (🟢) and then select the type of run as explained above.

A confirmation dialog box opens.

4-1-4. Click **Start Emulator and Run** to begin running the application.

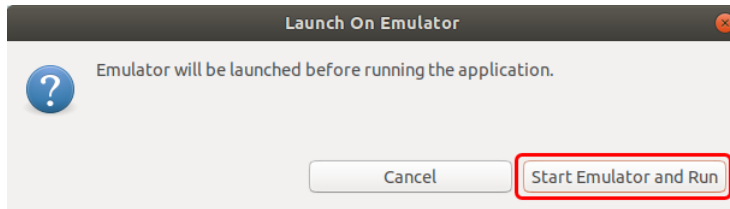


Figure 8-25: Launching the Emulator and Running the Application

Important: Sometimes, especially for Versal devices, the emulator does not launch and run cleanly. The workaround is to debug using the emulator. Once the debugging session begins, it can be "resumed", and the behavior will be the same as if it were run except that the behavior is more consistent.

Testing the Application

Step 5

The application should now be running. Test the operation of the application.

5-1. **Verify proper stopwatch operation.**

Time is displayed in mm.ss.milliseconds format.

The UART input (input from the terminal/console) for the specific functions (function → character to be sent to the UART from the keyboard) are:

- **GO → g**
- **PAUSE → p**
- **STOP and RESET → S**
- **RESET while running → r**

5-1-1. Press the START button and verify that the time value is incrementing.

Emulator users: Press <g> and then <Enter>.

5-1-2. Press the PAUSE button and verify that the time value stops incrementing.

Emulator users: Press <p> and then <Enter>.

5-1-3. Press the RESET button and verify that the time value resets to zero.

Emulator users: Press <r> and then <Enter>.

5-1-4. Repeat in any combination.

Does the stopwatch behave as expected?

Emulator users: Sometimes QEMU does not start properly, so you may have to restart QEMU to get the application to run.

5-2. Close the Vitis IDE.

5-2-1. Select **File** > **Exit** to save the 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 want to exit.

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

5-2-2. Click **Yes** to exit the tool if this dialog box appears.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/SWinterrupts` directory.

5-3. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

5-3-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$TRAINING_PATH/SWinterrupts`.

5-3-2. Select **SWinterrupts**.

5-3-3. Press <**Delete**>.

-- OR --

Using the command line:

5-3-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

5-3-5. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $TRAINING_PATH/SWinterrupts`

[Linux users]: `rm -rf $TRAINING_PATH/SWinterrupts`

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 leveraged the documentation on *scu_gic* and *ttcps* to locate the necessary information to complete this task. Finally, you verified the operation of the code on hardware.

Answers

- Fill in the names of the function calls that you will make to access the items marked A through E in the figure below.

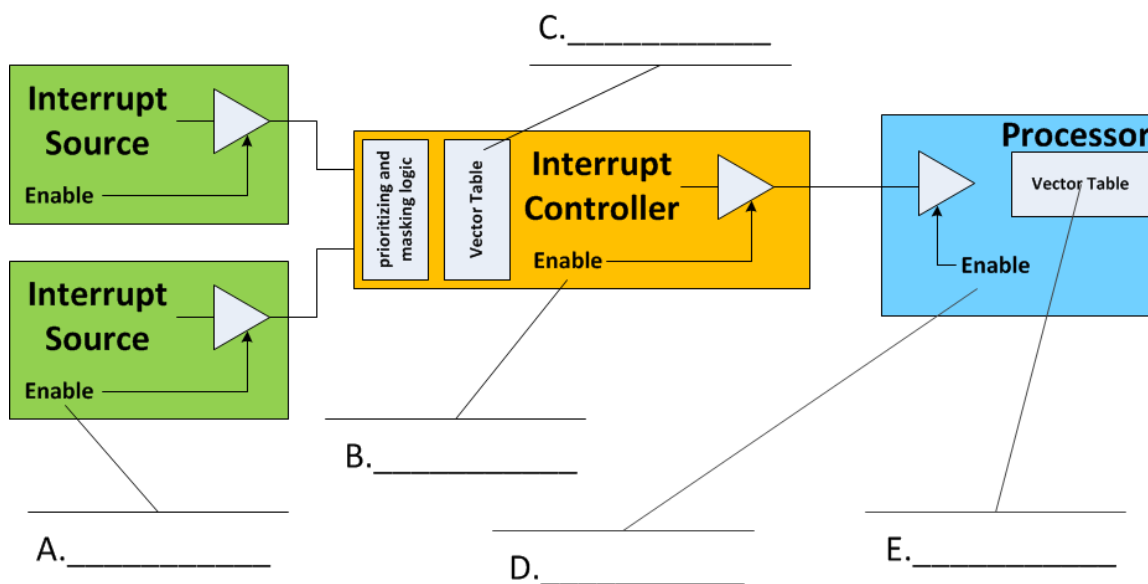


Figure 8-26: What Functions are Called for the Items Marked A-E?

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 device contains a number of timers and the one used here is the triple timer counter (TTC). The required function call is `XTtcPs_EnableInterrupts()`.

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 Zynq devices, the function used to enable any interrupt to the processor is `XScuGic_Enable()`.

C:

The vector table maps specific handlers to each interrupt. Typically there is one call per interrupt source.

The GIC in the Zynq devices uses *XScuGic_Connect()* to map the handler to the proper entry in the vector table.

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 as 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 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 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.