

Developing Zynq Software with Xilinx SDK

Lab 10 Interrupts



October 2013
Version 03

Lab 10 Overview

In systems programming an interrupt handler, also known as an Interrupt Service Routine (ISR), is a callback subroutine in microcontroller firmware, operating system, or device driver whose execution is triggered by the reception of a hardware interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the interrupt handler completes its task.

An interrupt handler is a low-level counterpart of event handlers. These handlers are typically initiated by hardware interrupts, and are used for servicing hardware devices and transitions between protected modes of operation such as system calls.

In Lab 8, we used the SDK Import feature to import some existing code into a blank application project. We will perform the same steps in this lab but only to start from a known good code base to access a piece of custom hardware built into the system. From that point, we will modify the application code to respond to an interrupt generated by the custom hardware platform when an incorrect PWM value is written to the PWM controller.

The LEDs referred to in this lab are implemented within the ZedBoard Programmable Logic design to connect the user PL LEDs (LD0-LD7) up to the PWM IP block.

Lab 10 Objectives

When you have completed Lab 10, you will know:

- How to enable the interrupt subsystem to allow hardware interrupts to interrupt software execution
- Create an interrupt service routine to handle the hardware interrupt

Experiment 1: Create the Interrupt Application Project

Similar to the flow for importing existing application code in Lab 8, a new blank project is created and existing code is imported. The application code is then executed to verify the application code we were given functions as expected.

Experiment 1 General Instruction:

Create a new blank software application project. Import code from the following folder:

C:\Speedway\ZynqSW\2013_3\Support_documents\LED_Dimmer

Run the application on the target hardware and observe the behavior.

Experiment 1 Step-by-Step Instructions:

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



Figure 1 – The SDK Application Icon

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

C:\Speedway\ZynqSW\2013_3\SDK_Workspace

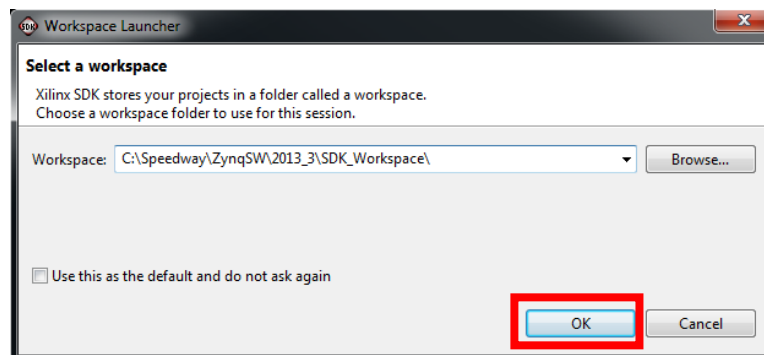


Figure 2 – Switching to the Appropriate SDK Workspace

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

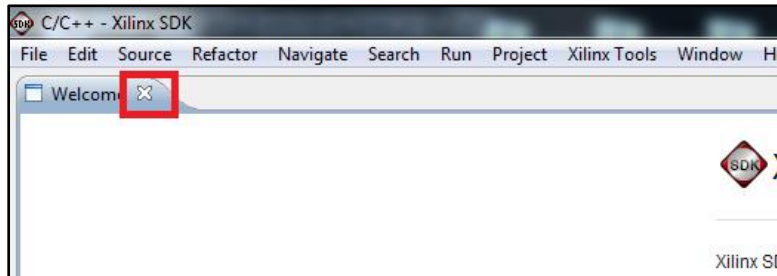


Figure 3 – Closing the SDK Welcome screen

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

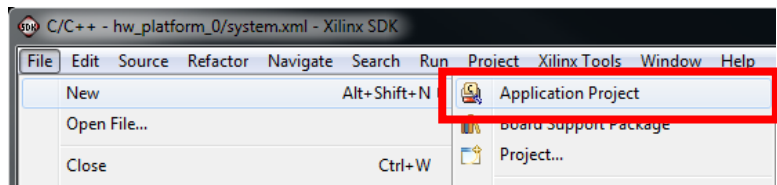


Figure 4 – Creating a New C Application Project

5. In the **New Project** wizard, change the **Project name** field to the **LED_Dimmer_w_Int** name.

Change the **Board Support Package** to the **Use existing** option and use the drop down menu to select the existing BSP **standalone_bsp_0** option.

Leave the other settings to their default values. Click the **Next** button to continue.

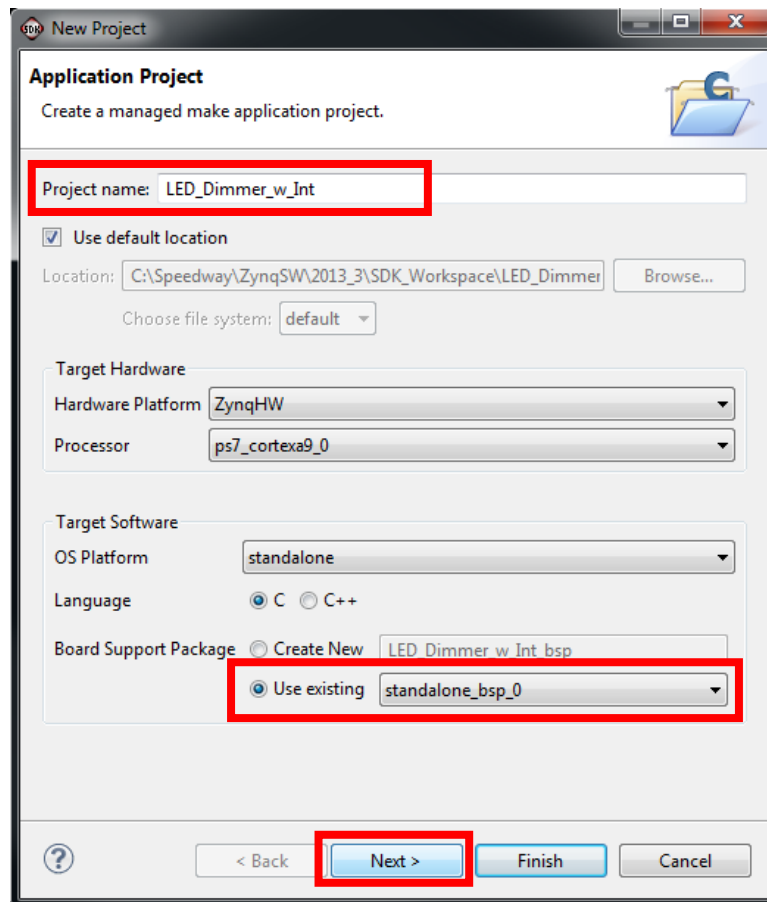


Figure 5 – Creating the LED_Dimmer_w_Int Application

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

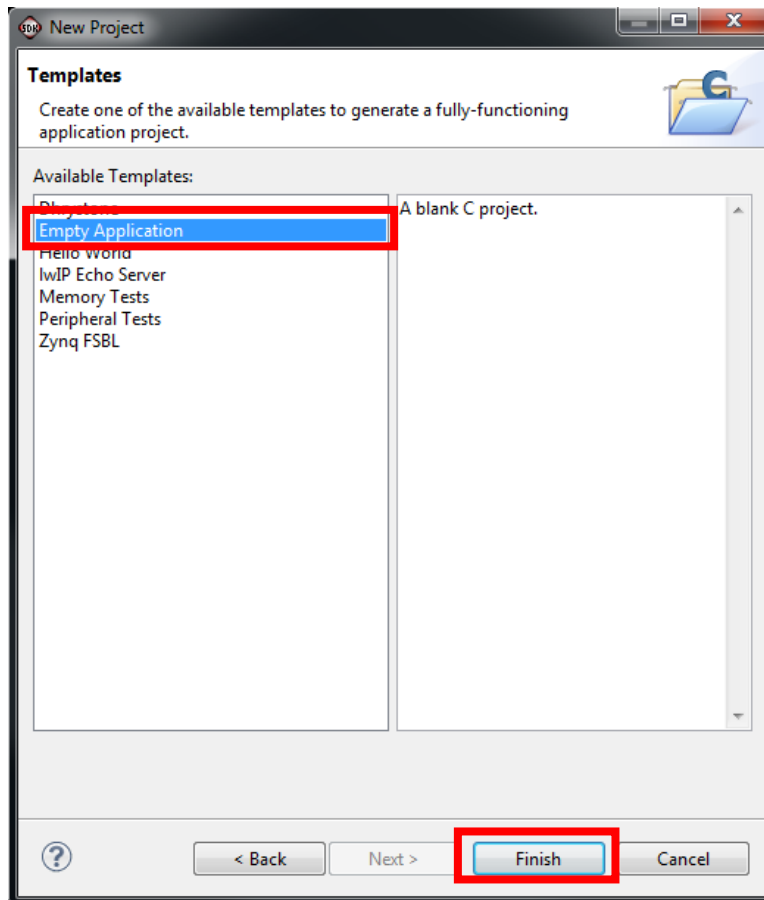


Figure 6 – Using the Empty Application Project Template

7. The empty **LED_Dimmer_w_Int** application project is created.

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

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

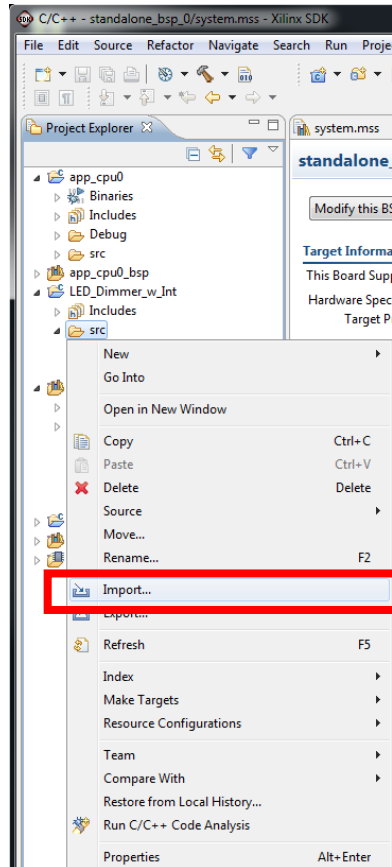


Figure 7 – Import LED_Dimmer Application Source Code

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

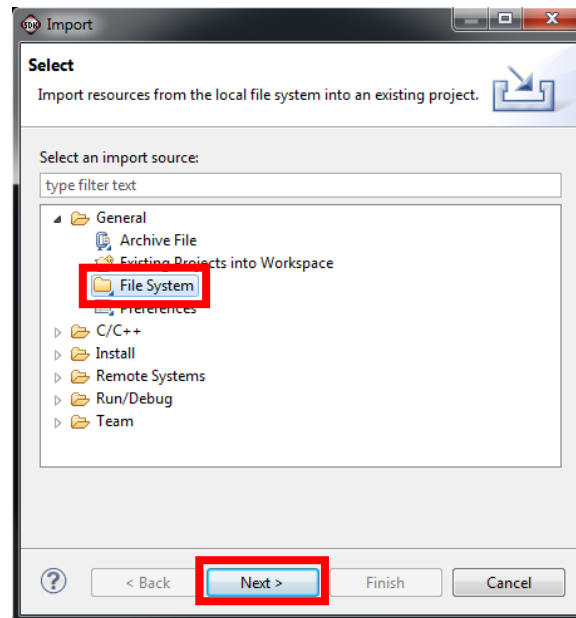


Figure 8 – Importing from a File System

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

C:\Speedway\ZynqSW\2013_3\Support_documents\LED_Dimmer

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

Select the **main.c** file and then click the **Finish** button to complete the **Import** operation.

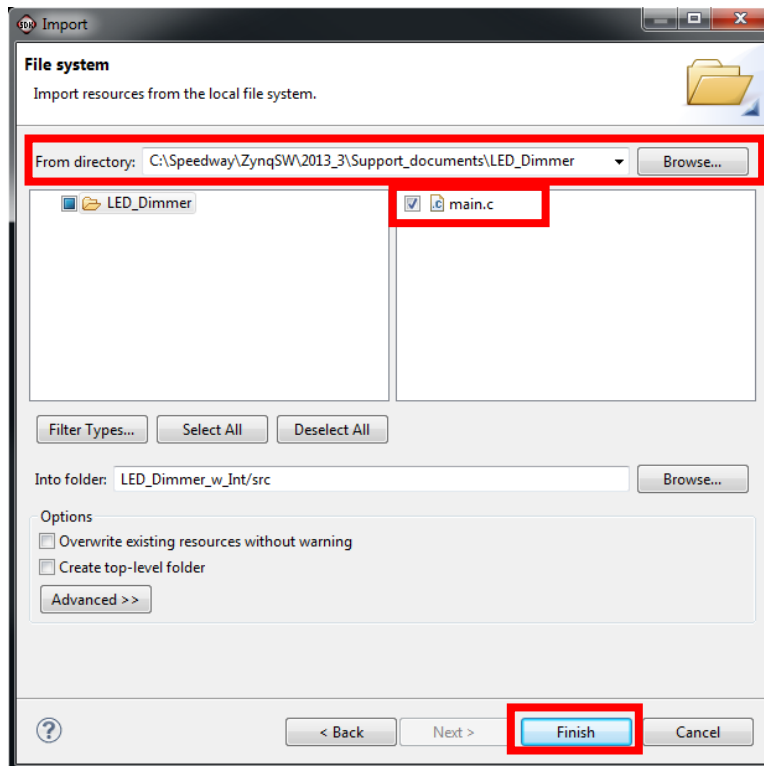



Figure 9 – Selecting Application Source Files

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



```
CDT Build Console [LED_Dimmer_w_Int]

'Building target: LED_Dimmer_w_Int.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T -Wl,../src/lscript.ld -L../app_cpu0_bsp/ps7_cortexa9_0/lib -o "LED_Dimmer_w_Int.elf" ./src/main.o
-Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: LED_Dimmer_w_Int.elf'

'Invoking: ARM Print Size'
arm-xilinx-eabi-size LED_Dimmer_w_Int.elf |tee "LED_Dimmer_w_Int.elf.size"
text      data      bss      dec      hex filename
22616     1096     29780    53492    d0f4 LED_Dimmer_w_Int.elf
'Finished building: LED_Dimmer_w_Int.elf.size'

14:33:58 Build Finished (took 428ms)
```

Figure 10 – Application Build Console Window

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

C:\Speedway\ZynqSW\2013_3\SDK_Workspace\LED_Dimmer_w_Int\Debug\LED_Dimmer_w_Int.elf

12. At this point the application is ready to be launched on the target hardware. Set the Boot Mode jumpers to Cascaded JTAG Mode, with JP7 through JP11 in the GND position.




Figure 11 – Cascaded JTAG Boot Mode

13. Attach another Micro-USB cable to the USB-UART port (J17) and make sure that the other end of this cable is connected to the development PC. For this example, the on-board USB-JTAG port will be used however the Digilent HS2 USB-JTAG cable or Xilinx Platform Cable USB II should work in a similar fashion through a connection to the PC4 header J15.



Figure 12 – Connecting On-board USB-JTAG Port

14. Turn power switch (SW8) to the ON position. ZedBoard will power on and the Green Power Good LED (LD13) should illuminate.
15. If necessary, launch or re-connect Tera Term.
16. In SDK, select **Xilinx Tools** → **Program FPGA** or click the  icon.

17. SDK will already know the correct .bit file (and .bmm if your future hardware platform includes that) since this was imported with the hardware platform. Click the **Program** button.

When LD12 lights blue, the PL has configured successfully. Look for the message “FPGA configured successfully with bitstream” in the Console window.

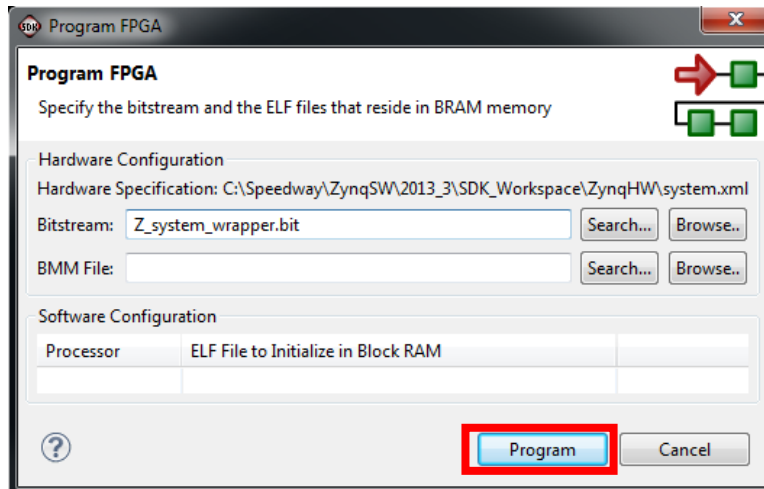


Figure 13 – Configure the Zynq Programmable Logic

18. In the **Project Explorer** tab, right-click on the **LED_Dimmer_w_Int** project folder.

Click on the **Run As** → **Launch on Hardware (GDB)** option in the pop up menu.

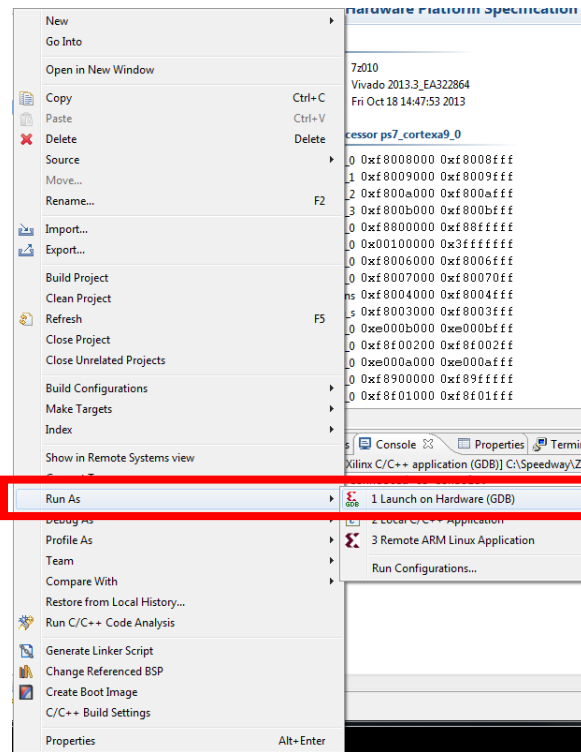


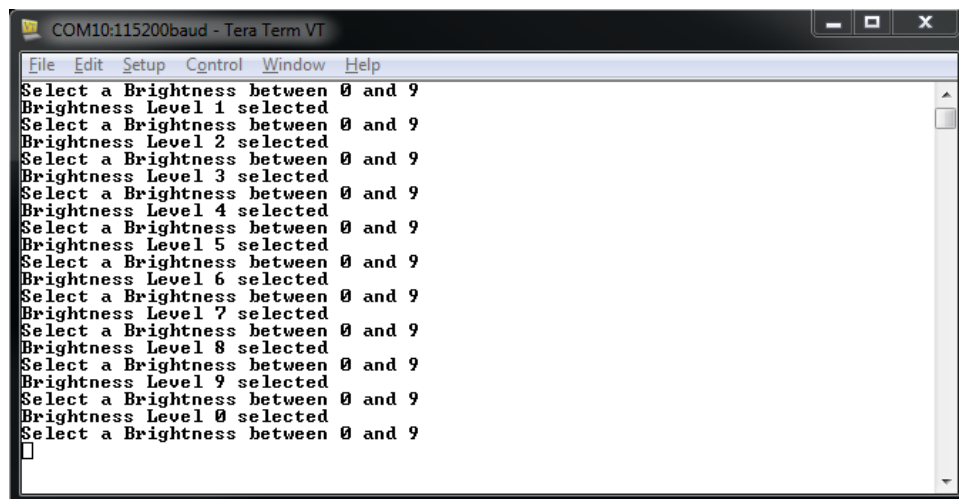
Figure 14 – Running LED_Dimmer_w_Int Application Code

19. Using the **Launch on Hardware** option will automatically create a run configuration using default settings and launch the application on the target platform. Once the application is launched and is running on the target hardware, a prompt should appear in the terminal window asking the user to **Select a Brightness between 0 and 9.**

20. Experiment with various inputs at the terminal to determine if the existing software application is sufficient for a user to assign a new brightness value to the PWM block.

One good check to make is to see how the application responds to an invalid input. Since we cannot enter more than one character at a time and the application is looking for an ASCII value from 0x30 to 0x39 to be entered, try entering in one of the alphabetical keys from A-Z to trigger an input outside of the valid range. Does the application respond the way you think it should to invalid input?

Since ZedBoard is used as a full development board for these course lab activities, the pre-built hardware platform does have the output of the PWM block tied to the Programmable Logic user LEDs LD0-LD7. By entering in different values within the range 0 to 9, you can view how the brightness value assigned to the core does indeed change the duty cycle of the PWM block output and thus changing the visible brightness of the ZedBoard PL user LEDs LD0 through LD7.



```
COM10:115200baud - Tera Term VT
File Edit Setup Control Window Help
Select a Brightness between 0 and 9
Brightness Level 1 selected
Select a Brightness between 0 and 9
Brightness Level 2 selected
Select a Brightness between 0 and 9
Brightness Level 3 selected
Select a Brightness between 0 and 9
Brightness Level 4 selected
Select a Brightness between 0 and 9
Brightness Level 5 selected
Select a Brightness between 0 and 9
Brightness Level 6 selected
Select a Brightness between 0 and 9
Brightness Level 7 selected
Select a Brightness between 0 and 9
Brightness Level 8 selected
Select a Brightness between 0 and 9
Brightness Level 9 selected
Select a Brightness between 0 and 9
Brightness Level 0 selected
Select a Brightness between 0 and 9

```

Figure 15 – Terminal Output from LED_Dimmer_w_Int Application

Questions:

Answer the following questions:

- *What happens when a value outside of the requested range of 0-9 is entered?*

- *Are interrupts currently enabled in the example application provided?*

Experiment 2: Add Interrupt Support to the Application

You may have noticed that the application code that we have been given does not do a very good job of handling incorrect values being passed in. In fact, it blindly passes incorrect values on to the PWM hardware block. This problem could be solved in software by first checking the values before writing the value to the PWM hardware block.

What if the application requires us to pass a value that could be invalid to the PWM hardware block?

In this experiment, we will enable the GIC to pass the PWM hardware block interrupt to the processor and interrupt execution flow. We will also add an Interrupt Service Routine (ISR) which will handle the interrupt from the PWM hardware block and assign a known **safe** value to the PWM hardware block before continuing application execution.

This is accomplished by the use of the BSP hardware driver **scugic** API. The following API calls will be used to add interrupt support:

```
int XScuGic_CfgInitialize
    (XScuGic *InstancePtr,
     XScuGic_Config *ConfigPtr,
     u32 EffectiveAddr)
```

- CfgInitialize a specific interrupt controller instance/driver.
- The function initializes the field of XscuGic structure and the initial vector table with stub function calls.
- All interrupt sources are disabled when initialization occurs.

```
int XScuGic_Connect
    (XScuGic *InstancePtr,
     u32 Int_Id,
     Xil_InterruptHandler Handler,
     void *CallBackRef)
```

- Makes the connection between the Int_Id of the interrupt source and the associated handler that is to run when the interrupt is recognized.

```
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.


```
void XScuGic_SetPriorityTriggerType
    (XScuGic *InstancePtr,
     u32 Int_Id,
     u8 Priority,
     u8 Trigger)
```

- Sets the interrupt priority and trigger type for the specified IRQ source.

```
XScuGic_Config * XScuGic_LookupConfig (u16 DeviceId)
```

- Looks up the device configuration based on the unique device ID.
- A table contains the configuration info for each device in the system.

```
void XScuGic_InterruptHandler (XScuGic *InstancePtr)
```

- This function is the primary interrupt handler for the driver.
- It must be connected to the interrupt source such that it is called when an interrupt of the interrupt controller is active.
- It will resolve which interrupts are active and enabled and call the appropriate interrupt handler.

The following API calls are not used in our application but could be useful for use in a future application:

```
void XScuGic_Disable
    (XScuGic *InstancePtr,
     u32 Int_Id)
```

- Disables the interrupt source provided as the argument Int_Id.

```
void XScuGic_GetPriorityTriggerType
    (XScuGic *InstancePtr,
     u32 Int_Id,
     u8 *Priority,
     u8 *Trigger)
```

- Gets the interrupt priority and trigger type for the specified IRQ source.

```
int XScuGic_SelfTest (XScuGic *InstancePtr)
```

- Runs a self-test on the driver and the associated hardware device.
- This function clears all of the interrupt enable bits.
- All interrupts will be disabled when this function is complete.

Experiment 2 General Instruction:

Open the **LED_Dimmer_w_Int** application source code.

Create an interrupt service routine (ISR) to handle interrupts from the PWM hardware block.

Add application code to enable interrupts and register the ISR.

Experiment 2 Step-by-Step Instructions:

1. Expand the **LED_Dimmer_w_Int** → **src** folder in the **Project Explorer** panel. Double click the application source file **main.c** to open it in the code editor.

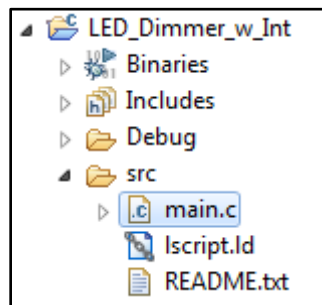


Figure 16 – LED_Dimmer_w_Int Source Code

2. In the source file we need to include the drivers for the **SCUGIC** and the **Exception Handler**. Place these lines below the other **#include** statements by inserting near line 49.

```
#include "xscugic.h"
#include "xil_exception.h"
```

3. Copy these definitions into the source code by inserting them near line 57 just below the existing definition for **PWM_BASE_ADDRESS**. These are static definitions that must be added to map our interrupt setup routine to the **SCUGIC** driver calls.

```
/* The following definitions are related to handling interrupts from the
 * PWM controller. */
#define XPAR_PS7_SCUGIC_0_DEVICE_ID 0
#define INTC_PWM_INTERRUPT_ID XPAR_FABRIC_PWM_W_INT_0_INTERRUPT_OUT_INTR
#define INTC XScuGic
#define INTC_HANDLER XScuGic_InterruptHandler
#define INTC_DEVICE_ID XPAR_PS7_SCUGIC_0_DEVICE_ID
```

4. Copy these variable declarations into the source code variable declarations section by inserting them near line 72 just within the **Variable Definitions** section. The **brightness** variable is now declared globally to make it visible to the ISR that will be added later. The **Intc** variable is a static definition used to setup the **SCUGIC** driver.

```
/* LED brightness level is now global to make is visible to the ISR. */
volatile u32 brightness;
/* The Instance of the Interrupt Controller Driver */
static INTC Intc;
```

5. Add this ISR into the source code following the variable declarations section by inserting it at line 77 just below the variable declarations inserted during Step 4 above. This will serve as the callback function that is called when the hardware interrupt is serviced.

```
void PWMIsr(void *InstancePtr)
{
    /* Inform the user that an invalid value was detected by the PWM
     * controller. */
    print("PWM Value exceeded, brightness reset to zero. Enter new value: \r\n");

    /* Set the brightness value to a safe value and write it to the
     * PWM controller in order to clear the pending interrupt. */
    brightness = 0;
    Xil_Out32(PWM_BASE_ADDRESS, brightness);
}
```

6. Add this setup function into the source code following the ISR section by inserting it near line 89 just below the ISR code inserted during Step 5 above. This code provides the setup needed to enable the PWM interrupt to be recognized and attaches the **PWMIsr()** as the interrupt handler.

```

/*****
/**
 * This function sets up the interrupt system for the PWM dimmer controller.
 * The processing contained in this function assumes the hardware system was
 * built with an interrupt controller.
 *
 * @param      None.
 *
 * @return      A status indicating XST_SUCCESS or a value that is contained in
 *              xstatus.h.
 *
 * @note       None.
 */
*****/
int SetupInterruptSystem()
{
    int result;
    INTC *IntcInstancePtr = &Intc;

    XScuGic_Config *IntcConfig;

    /* Initialize the interrupt controller driver so that it is ready to
     * use. */
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (IntcConfig == NULL)
    {
        return XST_FAILURE;
    }

    /* Initialize the SCU and GIC to enable the desired interrupt
     * configuration. */
    result = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                                   IntcConfig->CpuBaseAddress);
    if (result != XST_SUCCESS)
    {
        return XST_FAILURE;
    }

    XScuGic_SetPriorityTriggerType(IntcInstancePtr, INTC_PWM_INTERRUPT_ID,
                                   0xA0, 0x3);

    /* Connect the interrupt handler that will be called when an
     * interrupt occurs for the device. */
    result = XScuGic_Connect(IntcInstancePtr, INTC_PWM_INTERRUPT_ID,
                             (Xil_ExceptionHandler) PWMIsr, 0);
    if (result != XST_SUCCESS)
    {
        return result;
    }

    /* Enable the interrupt for the PWM controller device. */
    XScuGic_Enable(IntcInstancePtr, INTC_PWM_INTERRUPT_ID);

    /* Initialize the exception table and register the interrupt controller
     * handler with the exception table. */
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler) INTC_HANDLER, IntcInstancePtr);

    /* Enable non-critical exceptions */
    Xil_ExceptionEnable();

    return XST_SUCCESS;
}

```

7. Replace the **brightness** variable declaration near line 160 with this assignment to initialize **brightness** to a zero value. This variable is already declared globally so declaring it again in main would cause a compiler error.

```
brightness = 0;
```

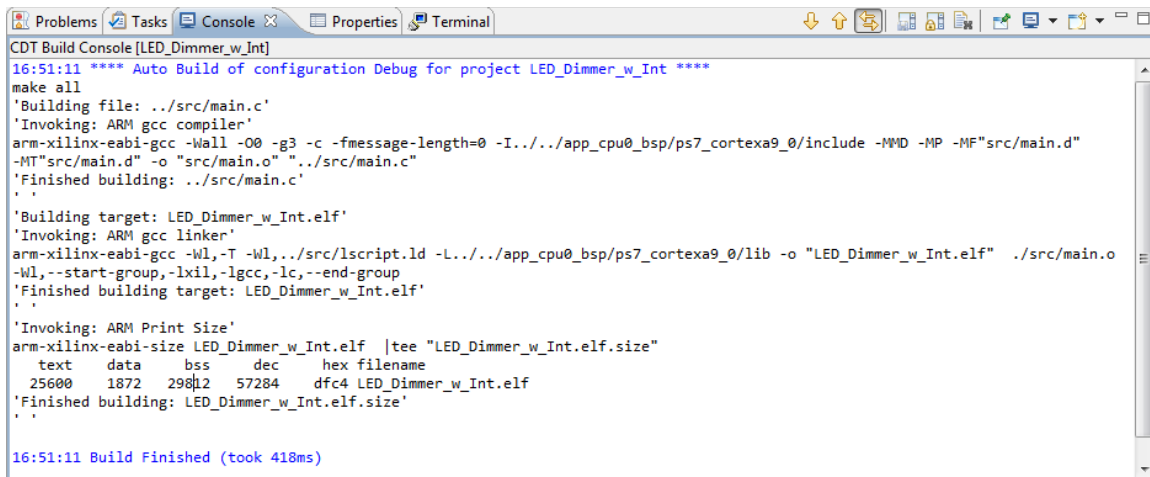
8. Add this call to **SetupInterruptSystem()** during our initialization steps within **main()** by inserting it near line 167 just below the following code:

```
Xil_Out32 (PWM_BASE_ADDRESS, 0) ;
```

This call will enable interrupts and register the ISR before any inputs from the user are processed.

```
/* Setup the interrupts such that interrupt processing can occur. If an
 * error occurs while setting up interrupts, then exit the application. */
status = SetupInterruptSystem();
if (status != XST_SUCCESS)
{
    return XST_FAILURE;
}
```

9. Save the updated source file. SDK will detect the source code change and automatically rebuild the application project and there should be no errors reported in the SDK console.



```
CDT Build Console [LED_Dimmer_w_Int]
16:51:11 **** Auto Build of configuration Debug for project LED_Dimmer_w_Int ****
make all
'Building file: ../src/main.c'
'Invoking: ARM gcc compiler'
arm-xilinx-eabi-gcc -Wall -O0 -g3 -c -fmessage-length=0 -I../app_cpu0_bsp/ps7_cortexa9_0/include -MMD -MP -MF"src/main.d"
-MT"src/main.d" -o "src/main.o" "../src/main.c"
'Finished building: ../src/main.c'
'Building target: LED_Dimmer_w_Int.elf'
'Invoking: ARM gcc linker'
arm-xilinx-eabi-gcc -Wl,-T ../src/lscrip.ld -L../app_cpu0_bsp/ps7_cortexa9_0/lib -o "LED_Dimmer_w_Int.elf" ../src/main.o
-Wl,--start-group,-lxil,-lgcc,-lc,--end-group
'Finished building target: LED_Dimmer_w_Int.elf'
'Invoking: ARM Print Size'
arm-xilinx-eabi-size LED_Dimmer_w_Int.elf |tee "LED_Dimmer_w_Int.elf.size"
text data bss dec hex filename
25600 1872 29812 57284 dfc4 LED_Dimmer_w_Int.elf
'Finished building: LED_Dimmer_w_Int.elf.size'
16:51:11 Build Finished (took 418ms)
```

Figure 17 – Rebuilding the Updated LED_Dimmer_w_Int Application

11. In the **Project Explorer** tab, right-click on the **LED_Dimmer_w_Int** project folder.

Click on the **Run As→ Launch on Hardware (GDB)** option in the pop up menu.

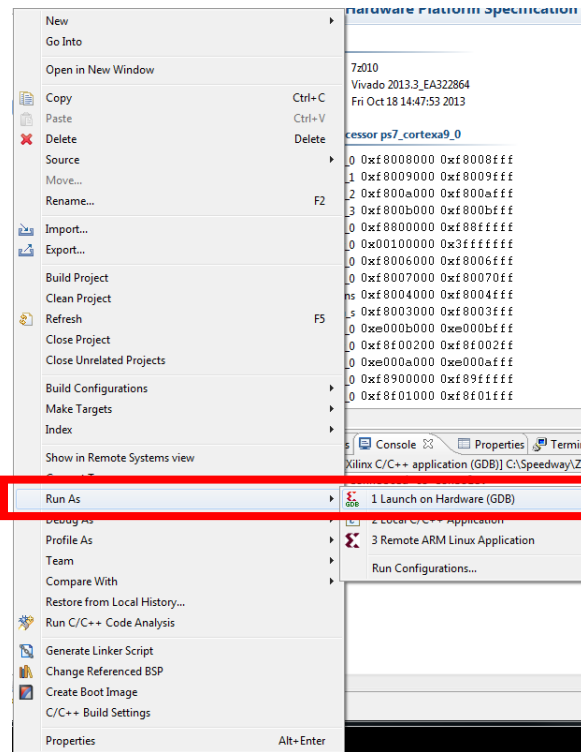


Figure 18 – Running LED_Dimmer_w_Int Application Code

12. If the application is still running on the target hardware from the previous experiment, a Processor in use warning will appear. Click on the **Yes** button to terminate the previous launch and run the updated application on the target hardware.

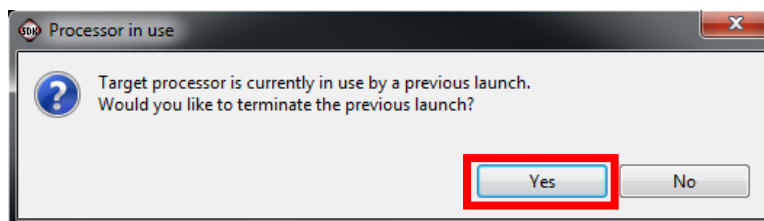
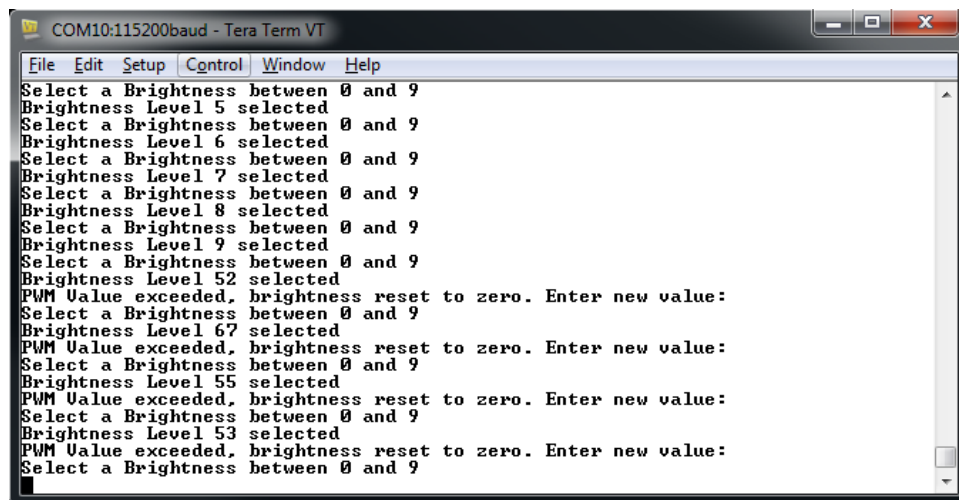


Figure 19 – Terminate Previous Launch

13. Once the application is launched and is running on the target hardware, a prompt should appear in the terminal window asking the user to **Select a Brightness between 0 and 9**.

Experiment with various inputs at the terminal to determine if the existing software application is any better suited for a user to assign a new brightness value to the PWM block.

One good check to make is to see how the application responds to an invalid input. Since we cannot enter more than one character at a time and the application is looking for an ASCII value from 0x30 to 0x39 to be entered, try entering in one of the alphabetical keys from A-Z to trigger an input outside of the valid range. Does the application respond the way you think it should to invalid input?



```
COM10:115200baud - Tera Term VT
File Edit Setup Control Window Help
Select a Brightness between 0 and 9
Brightness Level 5 selected
Select a Brightness between 0 and 9
Brightness Level 6 selected
Select a Brightness between 0 and 9
Brightness Level 7 selected
Select a Brightness between 0 and 9
Brightness Level 8 selected
Select a Brightness between 0 and 9
Brightness Level 9 selected
Select a Brightness between 0 and 9
Brightness Level 52 selected
PWM Value exceeded, brightness reset to zero. Enter new value:
Select a Brightness between 0 and 9
Brightness Level 67 selected
PWM Value exceeded, brightness reset to zero. Enter new value:
Select a Brightness between 0 and 9
Brightness Level 55 selected
PWM Value exceeded, brightness reset to zero. Enter new value:
Select a Brightness between 0 and 9
Brightness Level 53 selected
PWM Value exceeded, brightness reset to zero. Enter new value:
Select a Brightness between 0 and 9
```

Figure 20 – Terminal Output from LED_Dimmer_w_Int Application

Questions:

Answer the following questions:

- When does the function `PWMIsr()` get called?

- **GROUP DISCUSSION QUESTION** – Think about this and we will discuss when all participants are done with the labs.

We handled the ISR functionality inside the ISR, what is another way of doing this?

Exploring Further

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

- Add a global flag which gets set by the interrupt service routine and have the interrupt cleared by the main application loop once the flag is read.
- Remove the call to **SetupInterruptSystem()** and see if the interrupt service routine still gets called.

This concludes Lab 10.

Revision History

| Date | Version | Revision |
|-----------|---------|--------------------------------------|
| 12 Nov 13 | 01 | Initial Release |
| 25 Nov 13 | 02 | Revisions after pilot |
| 01 May 13 | 03 | ZedBoard.org Training Course Release |
| | | |

Resources

www.microzed.org

www.zedboard.org

www.em.avnet.com/drc

www.xilinx.com/zyng

www.xilinx.com/sdk

www.xilinx.com/vivado

www.xilinx.com/products/silicon-devices/soc/zyng-7000/ecosystem/index.htm

Answers

Experiment 1

- *What happens when a value outside of the requested range of 0-9 is entered?*

The value is accepted, multiplied by the brightness clock multiplier and assigned to the PWM hardware block.

- *Are interrupts currently enabled in the example application provided?*

Hardware interrupts are currently provided from the PWM hardware block which are then routed to the GIC in the hardware platform. However, the software application at this point does not configure the GIC to allow the application execution to be interrupted so the software end of the interrupt is not currently enabled.

Experiment 2

- *When does the function `PWMIsr()` get called?*

The **`PWMIsr()`** is the interrupt service routine callback function. It is registered with the interrupt subsystem as the callback interrupt handler through the call to **`XScuGic_Connect()`** during our interrupt setup.

- *GROUP DISCUSSION QUESTION – Think about this and we will discuss when all participants are done with the labs.*

We handled the ISR functionality inside the ISR, what is another way of doing this?

Alternatively, we could have the ISR just set a flag and then wait for the main function process the flag and write a message to the console.