# Lab 03 Report: Vivado AXI Interrupt

Peter Mowen
*peter.mowen@temple.edu*

**Summary**

Lab 03 introduces AXI interrupts. AXI interrupts are used to set the state of a finite state machine. Based on the state, different patterns are displayed on the LEDs and a count is incremented, decremented, or held.

## Introduction

The main objective of this lab is to review programming with interrupts. The secondary objective was to learn how to setup the interrupts for the Zynq chip. Zynq has exactly one interrupt pin. In this lab, exactly one interrupt is used so this is not a problem. Furthermore, the example code from *The Zynq Book Tutorials* configures the interrupt in software for the user.

The specifications for this lab, taken from the lab manual, are as follows (Silage, 2020):

a) BTN0 turns all the LEDs ON (1111) and pauses the current LED count
b) BTN1 increments the LED count by first 1 then 2 and repeat and outputs the count to the LEDs
c) BTN2 decrements the LED count by first 1 then 2 and repeat and outputs the count to the LEDs
d) BTN3 turns all the LEDs OFF (0000) and the count continues but not shown in the LEDs
e) If more than one BTN is depressed the LED count is paused and LED display is 1001 followed by 0110 and repeat
f) If no BTNs are depressed the LED count is incremented by 1 and the LED display continues

A finite state machine will be used to keep track of which buttons have been pressed. The interrupt service routine (ISR), will update an internal state variable. Based on the internal state, the LEDs will behave as per the specification.

## Discussion

<u>Hardware Setup in Vivado HLx</u>

The following screenshot shows the final hardware design used for this lab. It was generated by following the steps outlined in exercises 2A and 2B in *The Zynq Book Tutorials*.

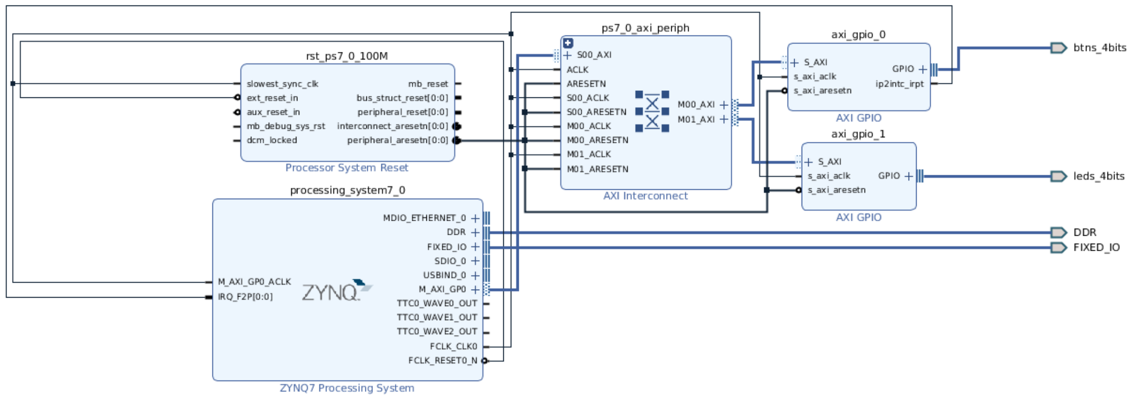


**Figure 1: Hardware Design in Vivado HLx**

This design is similar to the previous two labs: it connects two AXI GPIOs to the main processing system. However, there is a small difference if one inspects GPIO and the main processing block more closely:

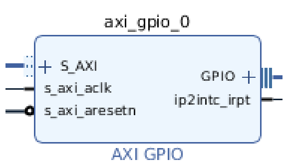


**Figure 2: Close-up of axi_gpio_0**

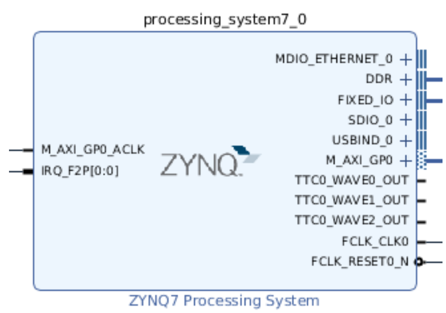


**Figure 3: Close-up of Processing System**

Note the pin *ip2ubtc_irpt* in Figure 2: Close-up of axi_gpio_0. This new pin is the interrupt output from GPIO 0. It connects to the interrupt input, *iIRQ_F2P[0:0]*, on the processing system, as seen in Figure 3: Close-up of Processing System. The system is now ready to use interrupts in the C code.

## Initializing GPIOs and Interrupts in C Code

In order to use interrupts, a few more Xilinx files are added to the include list as seen in the following screenshot:

```c
//----------------------------------------------------
// INCLUDE FILES
//----------------------------------------------------
#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

```

**Figure 4: Include files**

In addition to more include files, more parameter definitions are added are added:

```
//----------------------------------------------------
// PARAMETER DEFINTIONS
//----------------------------------------------------
#define INTC_DEVICE_ID         XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID         XPAR_AXI_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID         XPAR_AXI_GPIO_1_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define BTN_INT                XGPIO_IR_CH1_MASK
#define LED_DELAY              80000000    // Software delay length
#define printf xil_printf
#define DEBUG

```

**Figure 5: Parameter definitions**

The `INTC_DEVICE_ID` and `INTC_GPIO_INTERRUPT` parameters are new to this lab. The others have been used in labs one and two.

The follow is a screenshot of the prototype functions:

```
45 //----------------------------------------------------
46 // PROTOTYPE FUNCTIONS
47 //----------------------------------------------------
48 static void BTN_Intr_Handler(void *baseaddr_p);
49 static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
50 static int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr);
51 static int BoardInit();
52 static void UpdateLEDs();
```

**Figure 6: Prototype functions**

The prototype function on line 48 is for the button interrupt handler function. It takes a pointer to the memory location of the buttons. The next prototype function is for the interrupt system setup. It takes the pointer to the global interrupt controller. The function on line 50 is the interrupt controller initialization function. It takes the device ID and a pointer to the address of the GPIO instance. All of the above functions were supplied by *The Zync Book Tutorial*. The last two are written by this student and will be explained in detail later.

The following is a screenshot of the `BoardInt()` function, which is called from `main` to initialize the GPIOs and the interrupts:

```
221 int BoardInit()
222 {
223     int status;
224     //---------------------------------------------------
225     // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
226     //---------------------------------------------------
227 
228     // Initialize LEDs
229     status = XGpio_Initialize(&LEDInst, LEDS_DEVICE_ID);
230     if(status != XST_SUCCESS) return XST_FAILURE;
231 
232     // Initialize Push Buttons
233     status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
234     if(status != XST_SUCCESS) return XST_FAILURE;
235 
236     // Set LEDs direction to outputs
237     XGpio_SetDataDirection(&LEDInst, 1, 0x00);
238 
239     // Set all buttons direction to inputs
240     XGpio_SetDataDirection(&BTNInst, 1, 0xFF);
241 
242     // Initialize interrupt controller
243     status = IntcInitFunction(INTC_DEVICE_ID, &BTNInst);
244     if(status != XST_SUCCESS) return XST_FAILURE;
245 
246     return XST_SUCCESS;
247 }
```

**Figure 7: Function to initialize board**

Lines 223-240 are familiar at this point; the GPIOs for the LEDs and buttons are initialized and their data directions are set. The new code is line 243. This line calls a function to initialize the interrupt controller. It takes the interrupt controller device ID as its first argument and the address of the button instance as its second argument. This allows interrupts to be used throughout the program. A screenshot of this function can be found below:

```
265 int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr)
266 {
267     XScuGic_Config *IntcConfig;
268     int status;
269 
270     // Interrupt controller initialization
271     IntcConfig = XScuGic_LookupConfig(DeviceId);
272     status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
273     if(status != XST_SUCCESS) return XST_FAILURE;
274 
275     // Call to interrupt setup
276     status = InterruptSystemSetup(&INTCInst);
277     if(status != XST_SUCCESS) return XST_FAILURE;
278 
279     // Connect GPIO interrupt to handler
280     status = XScuGic_Connect(&INTCInst,
281                              INTC_GPIO_INTERRUPT_ID,
282                              (Xil_ExceptionHandler)BTN_Intr_Handler,
283                              (void *)GpioInstancePtr);
284     if(status != XST_SUCCESS) return XST_FAILURE;
285 
286     // Enable GPIO interrupts interrupt
287     XGpio_InterruptEnable(GpioInstancePtr, 1);
288     XGpio_InterruptGlobalEnable(GpioInstancePtr);
289 
290     // Enable GPIO and timer interrupts in the controller
291     XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
292 
293     return XST_SUCCESS;
294 }
```

**Figure 8: Interrupt Controller Initialization Function**

The code on line 271 creates an object which contains information about the interrupt controller. On the next line, this information is passed into a function that initializes the interrupt controller. This function takes as an input the address of the interrupt controller instance, the object created on line 271, and the CPU base address pulled from the interrupt controller configuration object. The status variable holds whether or not the processes succeeded or failed. This function calls the interrupt system setup function, pictured below.

```
249 int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
250 {
251     // Enable interrupt
252     XGpio_InterruptEnable(&BTNInst, BTN_INT);
253     XGpio_InterruptGlobalEnable(&BTNInst);
254
255     Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
256                                  (Xil_ExceptionHandler)XScuGic_InterruptHandler,
257                                  XScuGicInstancePtr);
258     Xil_ExceptionEnable();
259
260
261     return XST_SUCCESS;
262 }
```

**Figure 9: Interrupt System Setup function**

This function enables the button interrupt.  The code on line 255 registers instance of the interrupt controller with the systems exception handler. Then exceptions are enabled.

<u>Control Logic Design</u>

A finite state machine is used control the LEDs based on the state of the machine. The following is a screenshot of the global variables representing the machine states:

```
// button states
enum btn_state {NO_BTN, BTN0, BTN1, BTN2, BTN3, ERROR};
enum btn_state current_state;

// event states
enum event_state {FIRST, SECOND};
enum event_state error;     // which error code, 1 or 2
enum event_state increment; // whether to increment by 1 or 2
enum event_state decrement; // whether to decrement by 1 or 2
```

**Figure 10: States for the finite state machine**

The enumeration `btn_state` defines the states for the state machine. These correspond to the states of no buttons being pressed, button zero being pressed, button one being pressed, button two being pressed, button three being pressed, and finally, more than one button being pressed. The `event_states` will be explained shortly. These states are assigned according to the following finite state machine:

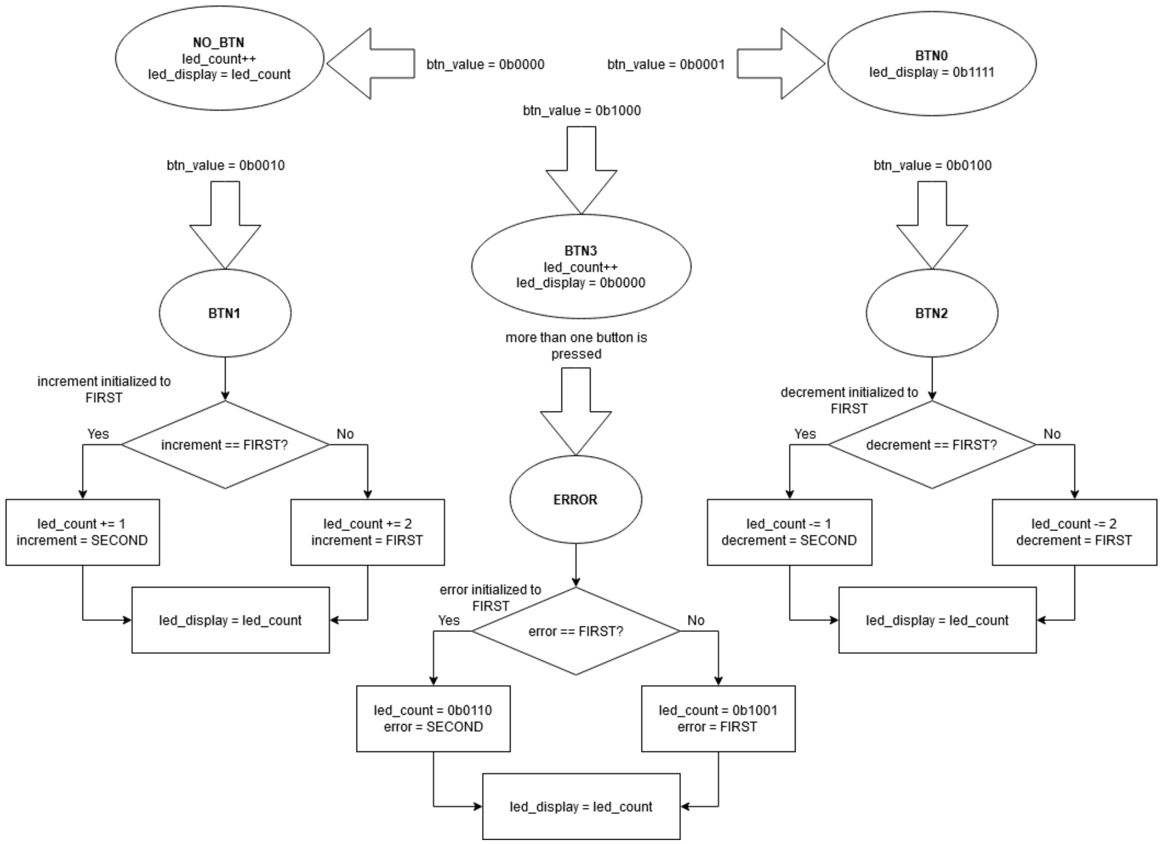


**Figure 11: Finite state machine**

The states, listed in bold, are set by the button interrupt service routine (ISR). This ISR is called when any button changes states. That is, if a button goes from not pressed to pressed or from pressed to not pressed. The following screenshots shows the first part of the ISR code:

```
120 //----------------------------------------------------
121 // Run on button press and release.
122 // Sets current_state based on pressed buttons
123 //----------------------------------------------------
124 void BTN_Intr_Handler(void *InstancePtr)
125 {
126     // Disable GPIO interrupts
127     XGpio_InterruptDisable(&BTNInst, BTN_INT);
128 
129     // Ignore additional button presses
130     if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) !=
131             BTN_INT) {
132             return;
133         }
134 
135     // debounce time. needed to wait to see if more than one button is pressed
136     for (int Delay = 0; Delay < 5000000; Delay++);
137 
138     // Read button value
139     btn_value = XGpio_DiscreteRead(&BTNInst, 1) & 0xF;
140 #ifdef DEBUG
141     printf("button value = %d\n", btn_value);
142 #endif
143 
```

**Figure 12: Button press ISR, part 1**

In the first part of the ISR, the button interrupt is disabled on line 127. The `if` statement on line 130 gets the status of the interrupt associated with the buttons and ANDs it with the button interrupt mask. If the result is not equal to the button interrupt mask, then the interrupt was not disabled and the ISR is exited. Then the code verifies that the interrupts are disabled on lines 130-133. Next, the code waits for a brief moment using an empty `for` loop. This `for` loop was necessary because the time between a single button being pressed and the value being read was faster than the time it took to press two buttons "simultaneously." Without that wait, every time two buttons are pressed, the ISR sets the current state to whichever one was pressed "first," then updates the state for two or more buttons being pressed. If this lab used timers, the code would have been structured so that the button press started a timer, then the state would have been set from the timer interrupt.

The next section of the ISR sets the machine state based on the value read in from the buttons:

```
144     // Update the current state based on button value
145     switch(btn_value)
146     {
147         case 0b0000:
148             current_state = NO_BTN;
149             break;
150         case 0b0001:
151             current_state = BTN0;
152             break;
153         case 0b0010:
154             current_state = BTN1;
155             break;
156         case 0b0100:
157             current_state = BTN2;
158             break;
159         case 0b1000:
160             current_state = BTN3;
161             break;
162         default:
163             current_state = ERROR;
164             break;
165     }
166
167     UpdateLEDs();
168
169     (void)XGpio_InterruptClear(&BTNInst, BTN_INT);
170
171     // Enable GPIO interrupts
172     XGpio_InterruptEnable(&BTNInst, BTN_INT);
173 }
```

**Figure 13: Button press ISR, part 2**

As can be see from the code, the state is set to correspond with which buttons have been pressed. The case where no buttons are pressed happens when all of the buttons have been released. Once the state is set, the `UpdateLEDs()` function is called, the GPIO interrupt flag is cleared, and interrupts are re-enabled.

The `UpdateLEDs()` function determines what to do based on the current state as shown in the following screenshots:

```c
//-----------------------------------------------------
// Update LEDs based on current_state set by interrupt
//-----------------------------------------------------
void UpdateLEDs()
{
#ifdef DEGBUG
    printf("Current state = %d\n", current_state);
#endif
    switch(current_state)
    {
        case NO_BTN:    // no buttons pressed
            // handled in main
            break;
        case BTN0:  // BTN0 pressed
            led_display = 0b1111;
            break;
        case BTN1:  // BTN1 pressed
            if (increment == FIRST)
            {// increment count by 1
                led_count += 1;
                increment = SECOND;
            }
            else if (increment == SECOND)
            {// increment count by 2
                led_count +=2;
                increment = FIRST;
            }
            led_display = led_count;
            break;
        case BTN2:  // BTN2 pressed
            if (decrement == FIRST)
            {// decrement count by 1
                led_count -= 1;
                decrement = SECOND;
            }
            else if (decrement == SECOND)
            {// decrement count by 2
                led_count -=2;
                decrement = FIRST;
            }
            led_display = led_count;
            break;
        case BTN3:  // BTN3 pressed
            // increment count...
            led_count++;
            // but don't show it on the LEDs
            led_display = 0;
            break;
        case ERROR: // More than one button pressed
            if (error == FIRST)
            {// First part of error code
                led_display = 0b1001;
                error = SECOND; // do the SECOND part next time
            }
            else if (error == SECOND)
            {// Second part of error code
                led_display = 0b0110;
                error = FIRST; // do the FIRST part next run
            }
            break;
    }

    // Write to LEDs
    XGpio_DiscreteWrite(&LEDInst, 1, led_display);
}
```

**Figure 14: `UpdateLEDs()` function code**

The `NO_BTN` state is handled in `main` and will be discussed later. If the state is set to `BTN0`, the `led_display` is set to all ones and nothing is done to the `led_count`. When the state is `BTN1`, the event variable `increment` is checked. If it is set to `FIRST`, then `led_count` is incremented by one; else, if it is set to `SECOND`, then `led_count` is incremented by two. Once the `led_count` is updated, `led_display` is set to `led_count`. A similar action happens for the `BTN2` state, but the count is decremented by one or two based on the `decrement` variable. If the state is `BTN3`, then `led_count` is incremented and `led_display` is set to zero. Finally, if the state is `ERROR`, the error event state variable is checked. If it is `FIRST`, then the `led_display` is set to 1001 and if it is set to `SECOND`, `led_display` is set to 0110. Finally, the `led_display` is written to the LEDs.

The following is a screenshot of the `main` function:

```c
//---------------------------------------------------
// MAIN FUNCTION
//---------------------------------------------------
int main (void)
{
    // Initialize board
    int status = BoardInit();
    if (status != XST_SUCCESS) return status;

    // Declare local Delay variable
    int Delay;

    // Initialize led_count and led_display
    led_count = 0b0000;
    led_display = 0b0000;

    // Initialize button state
    current_state = NO_BTN;

    // Initialize event states
    increment = FIRST;
    decrement = FIRST;
    error = FIRST;

    while(1)
    {
        // Update led_count and led_display based on state and write to LEDs
        UpdateLEDs();

        // Wait about one second for user to be able to read LEDs
        for (Delay = 0; Delay < LED_DELAY; Delay++);

        // Update led variables for when no buttons are pressed
        if (current_state == NO_BTN)
        {// this gets written out to LEDs in UpdateLEDs
            led_count++;
            led_display = led_count;
        }
    }
    return 0;
}
```

**Figure 15: `main` function**

First, the `BoardInit()` function is called to initialize the hardware and interrupts as previously discussed. Next, the `Delay`, which is used in the `for` loop that causes the delay, is declared. Next `led_count` and `led_display` are initialized to zero. Next, each of the event state variables are initialized to `FIRST`. Finally, the infinite `while` loop is entered. This loop is simple. First the leds are updated, then it waits for about one second, then state `NO_BTN` is handled.

Something to note: it is possible that the interrupt would be called while `UpdateLEDs()` is running from main. However, this is extremely unlikely. The time the program spends in `UpdateLEDs()` is much less than the time it is in the empty `for` loop or handling the `NO_BTN` state.

Results
As can be seen in the photos below, the LEDs behaved as expected for BTN0, BTN3 and more than one button being pressed.

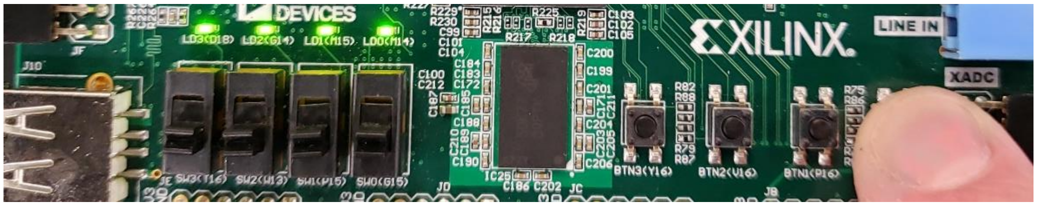


**Figure 16: BTN0 being pressed**

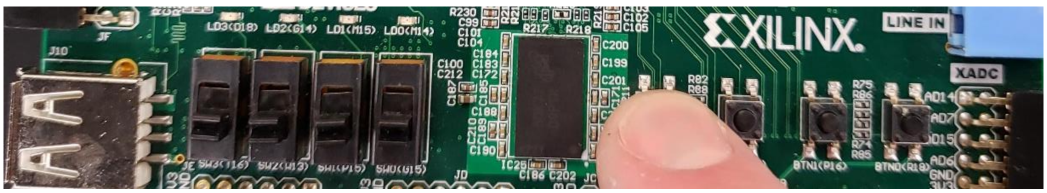


**Figure 17: BTN3 being pressed**


**Figure 18: Two buttons being pressed at the same time**

Also, a full demonstration of this lab can be found in the following YouTube video:

https://youtu.be/g0aSq0J1obI

## Conclusions

The specifications for this lab were completed without polling the buttons. Therefore, the concept how to use an interrupt was successfully introduced. However, how to setup the interrupts was not well grasped. By using the example code from *The Zynq Book Tutorials*, this student did not spend the time to learn how setting up the interrupt worked in software. The supporting material in *The Zynq Book Tutorials* was vague on the details of what each function in the initialization functions did.

## Appendix

References

Louise H. Crocket, R. A. (2015). *The Zynq Book Tutorials for Zybo and ZedBoard.* Glasgow, Scotland, UK: University of Strathclyde.

Silage, D. (2020). *Vivado AXI Interrupt.*

C Code

```
//----------------------------------------------------
// INCLUDE FILES
//----------------------------------------------------
#include "xparameters.h"
#include "xgpio.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

//----------------------------------------------------
// PARAMETER DEFINTIONS
//----------------------------------------------------
#define INTC_DEVICE_ID            XPAR_PS7_SCUGIC_0_DEVICE_ID
#define BTNS_DEVICE_ID            XPAR_AXI_GPIO_0_DEVICE_ID
#define LEDS_DEVICE_ID            XPAR_AXI_GPIO_1_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define BTN_INT                   XGPIO_IR_CH1_MASK
#define LED_DELAY                 80000000     // Software delay length
#define printf xil_printf
#define DEBUG

//----------------------------------------------------
// OBJECT DELCARATIONS
//----------------------------------------------------
XGpio LEDInst, BTNInst;
XScuGic INTCInst;

//----------------------------------------------------
// GLOBAL VARIABLES
//----------------------------------------------------
static int led_count;      // count to be displayed
static int led_display;    // what gets written to LEDs
static int btn_value;      // lowest four bits contain button state

// button states
enum btn_state {NO_BTN, BTN0, BTN1, BTN2, BTN3, ERROR};
enum btn_state current_state;

// event states
enum event_state {FIRST, SECOND};
enum event_state error;           // which error code, 1 or 2
enum event_state increment;       // whether to increment by 1 or 2
enum event_state decrement;       // whether to decrement by 1 or 2

//----------------------------------------------------
// PROTOTYPE FUNCTIONS
//----------------------------------------------------
static void BTN_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr);
```

```c
static int BoardInit();
static void UpdateLEDs();

//--------------------------------------------------
// Update LEDs based on current_state set by interrupt
//--------------------------------------------------
void UpdateLEDs()
{
#ifdef DEGBUG
	printf("Current state = %d\n", current_state);
#endif
	switch(current_state)
	{
		case NO_BTN: // no buttons pressed
			// handled in main
			break;
		case BTN0:   // BTN0 pressed
			led_display = 0b1111;
			break;
		case BTN1:   // BTN1 pressed
			if (increment == FIRST)
			{// increment count by 1
				led_count += 1;
				increment = SECOND;
			}
			else if (increment == SECOND)
			{// increment count by 2
				led_count +=2;
				increment = FIRST;
			}
			led_display = led_count;
			break;
		case BTN2:   // BTN2 pressed
			if (decrement == FIRST)
			{// decrement count by 1
				led_count -= 1;
				decrement = SECOND;
			}
			else if (decrement == SECOND)
			{// decrement count by 2
				led_count -=2;
				decrement = FIRST;
			}
			led_display = led_count;
			break;
		case BTN3:   // BTN3 pressed
			// increment count...
			led_count++;
			// but don't show it on the LEDs
			led_display = 0;
			break;
		case ERROR:  // More than one button pressed
			if (error == FIRST)
			{// First part of error code
				led_display = 0b0110;
				error = SECOND;     // do the SECOND part next time
			}

			else if (error == SECOND)
			{// Second part of error code
				led_display = 0b1001;
				error = FIRST; // do the FIRST part next run
			}
```

```
			break;
	}


	// Write to LEDs
	XGpio_DiscreteWrite(&LEDInst, 1, led_display);
}

//----------------------------------------------------
// Run on button press and release.
// Sets current_state based on pressed buttons
//----------------------------------------------------
void BTN_Intr_Handler(void *InstancePtr)
{
	// Disable GPIO interrupts
	XGpio_InterruptDisable(&BTNInst, BTN_INT);

	// Ignore additional button presses
	if ((XGpio_InterruptGetStatus(&BTNInst) & BTN_INT) !=
			BTN_INT) {
			return;
		}

	// debounce time. needed to wait to see if more than one button is pressed
	for (int Delay = 0; Delay < 5000000; Delay++);

	// Read button value
	btn_value = XGpio_DiscreteRead(&BTNInst, 1) % 0xF;
#ifdef DEBUG
	printf("button value = %d\n", btn_value);
#endif

	// Update the current state based on button value
	switch(btn_value)
	{
		case 0b0000:
			current_state = NO_BTN;
			break;
		case 0b0001:
			current_state = BTN0;
			break;
		case 0b0010:
			current_state = BTN1;
			break;
		case 0b0100:
			current_state = BTN2;
			break;
		case 0b1000:
			current_state = BTN3;
			break;
		default:
			current_state = ERROR;
			break;
	}

	UpdateLEDs();


	(void)XGpio_InterruptClear(&BTNInst, BTN_INT);

    // Enable GPIO interrupts
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
}
```

```
//--------------------------------------------------
// MAIN FUNCTION
//--------------------------------------------------
int main (void)
{
	// Initialize board
	int status = BoardInit();
	if (status != XST_SUCCESS) return status;

	// Declare local Delay variable
	int Delay;

	// Initialize led_count and led_display
	led_count = 0b0000;
	led_display = 0b0000;

	// Initialize button state
	current_state = NO_BTN;

	// Initialize event states
	increment = FIRST;
	decrement = FIRST;
	error = FIRST;

	while(1)
	{
		// Update led_count and led_display based on state and write to LEDs
		UpdateLEDs();

		// Wait about one second for user to be able to read LEDs
		for (Delay = 0; Delay < LED_DELAY; Delay++);

		// Update led variables for when no buttons are pressed
		if (current_state == NO_BTN)
		{// this gets written out to LEDs in UpdateLEDs
			led_count++;
			led_display = led_count;
		}
	}
	return 0;
}

//--------------------------------------------------
// INITIAL SETUP FUNCTIONS
//--------------------------------------------------

int BoardInit()
{
	int status;
	//--------------------------------------------------
	// INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
	//--------------------------------------------------

	// Initialize LEDs
	status = XGpio_Initialize(&LEDInst, LEDS_DEVICE_ID);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Initialize Push Buttons
	status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Set LEDs direction to outputs
```

```
	XGpio_SetDataDirection(&LEDInst, 1, 0x00);

	// Set all buttons direction to inputs
	XGpio_SetDataDirection(&BTNInst, 1, 0xFF);

	// Initialize interrupt controller
	status = IntcInitFunction(INTC_DEVICE_ID, &BTNInst);
	if(status != XST_SUCCESS) return XST_FAILURE;

	return XST_SUCCESS;
}

int InterruptSystemSetup(XScuGic *XScuGicInstancePtr)
{
	// Enable interrupt
	XGpio_InterruptEnable(&BTNInst, BTN_INT);
	XGpio_InterruptGlobalEnable(&BTNInst);

	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler)XScuGic_InterruptHandler, XScuGicInstancePtr);
	Xil_ExceptionEnable();


	return XST_SUCCESS;
}

int IntcInitFunction(u16 DeviceId, XGpio *GpioInstancePtr)
{
	XScuGic_Config *IntcConfig;
	int status;

	// Interrupt controller initialization
	IntcConfig = XScuGic_LookupConfig(DeviceId);
	status   =   XScuGic_CfgInitialize(&INTCInst,   IntcConfig,   IntcConfig-
>CpuBaseAddress);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Call to interrupt setup
	status = InterruptSystemSetup(&INTCInst);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Connect GPIO interrupt to handler
	status = XScuGic_Connect(&INTCInst,INTC_GPIO_INTERRUPT_ID,
					(Xil_ExceptionHandler)BTN_Intr_Handler,
					(void *)GpioInstancePtr);
	if(status != XST_SUCCESS) return XST_FAILURE;

	// Enable GPIO interrupts interrupt
	XGpio_InterruptEnable(GpioInstancePtr, 1);
	XGpio_InterruptGlobalEnable(GpioInstancePtr);

	// Enable GPIO and timer interrupts in the controller
	XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);

	return XST_SUCCESS;
}
```