Lab 3 – Vivado AXI Timer and Interrupts

Leomar Durán

Summary

Introduction

In step 2, the specification details adding the reset switch SW0. However, it only specifies what to do when SW0 is on. I assumed that when SW0 is off, the correct behavior would be to re-enable the button interrupts. The LED count display will start up on its own again, and the number of expiration events can be updated without further action.

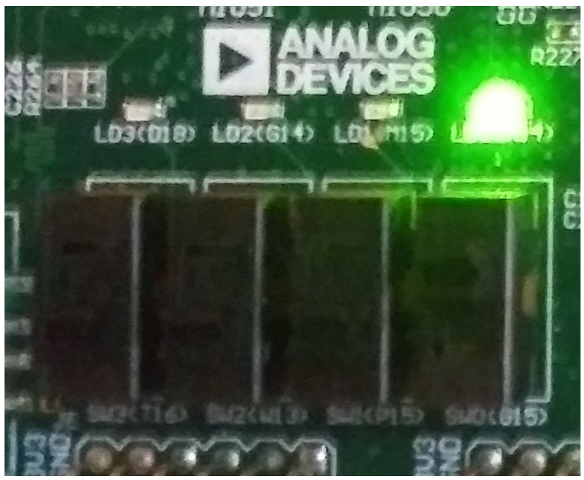

Figure 1: The LED count displays 0b0001 after SW0 is thrown because the timer interrupt increases it before updating it.

Also, when SW0 reset the LED count, the timer is still updating the display, and it increases before it updates. So the display gets 0b0001, as seen in Fig. 1 instead of 0b0000. I assumed that this is the correct behavior. An alternative solution would be to disable the timer interrupts as well. Another alternative would be for SW0 to set the LED count to 0b1111, which will result in a display of 0b0000.

In step 3, since it is necessary to reconfigure the timer for button debounce to do this step, first the project was tested with DEFAULT_N_EXPIRES at 3, which results in $41.71\,\mathrm{s}$ for 32 LED counts. Then it was tested with DEFAULT_N_EXPIRES at 7, which results in $43.58\,\mathrm{s}$ for 32 LED counts. This gives a rate of difference of $\frac{2|43.58\,\mathrm{s}-41.71\,\mathrm{s}|}{43.58\,\mathrm{s}-41.71\,\mathrm{s}} = 0.04385$.

Thus, I assumed that the implementation for `TMR_Intr_Handler(void *):void` was not correct. In fact, when the timer expires, it is not reset and restarted until counting to n_expires which does not perform any noticeable delay.

Now lowering TMR_LOAD causes the delay between switches to rise, so the timer is assumed to be in normal mode.

Also the specification states to reconfigure `BTN1` so that it increments the number of expiration events, and stops after a maximum of 7. It does not say what to do after that. So it was let to cascade down to its original behavior, that is, to increase the LED count display by its value `0b0010`.

Discussion

interrupt controller tut 2D.c

The unmodified *interrupt_controller_tut_2D.c* project has five module functions. Namely, these are:

- `BTN_Intr_Handler(void *):void`,
- `TMR_Intr_Handler(void *):void`,
- `main(void):int`,
- `InterruptSystemSetup(XScuGic *):int`, and
- `IntcInitFunction(u16, XTmrCtr *, XGpio *):int`.

BTN Intr Handler

The `BTN_Intr_Handler(void *):void` module function handles button interrupts by increasing the LED counter by the buttons pressed. Specifically, it performs Algorithm 1.

Algorithm 1: BTN INTR HANDLER

Input: unused instance pointer.
DISABLE button interrupts on `axi_gpio_0`.
if *there are other interrupts from* `axi_gpio_0` *not from channel* 1 **then**
    return to the caller.
READ in the value of the button.
ADD the value of the button to the LED counter data.
WRITE the LED counter data to channel 1 on `axi_gpio_1`.
CLEAR the button interrupt flag on `axi_gpio_0`.
RE-ENABLE button interrupts on `axi_gpio_0`.

TMR Intr Handler

The `TMR_Intr_Handler(void *):void` module function handles timer interrupts, by periodically incrementing the LED counter. Specifically it performs the Algorithm 2.

Algorithm 2: TMR INTR HANDLER

Input: unused data pointer
if *the timer expires* **then**
  if *this is the expiration #3* **then**
    STOP the timer counter.
    RESET the expiration count.
    INCREMENT the LED counter data.
    WRITE the LED counter data to channel 1 on `axi_gpio_1`.
    RESET the timer counter.
    START the timer counter.
  else
    INCREMENT the expiration count.

Main

The `main(void):int` module function orchestrates all of the operations necessary to run the program. In this case, it initializes the peripheral devices (the LEDs and buttons) and the corresponding GPIOs. It also starts the timer and initially enables the interrupts.

Then it polls.

Algorithm 3: MAIN

Output: 0 on success; `XST_FAILURE` if there is either an error initializing the peripherals or the timer.

INITIALIZE the instance for `api_gpio_1`, the GPIO for LEDs.
if *the initialization was not successful* **then**
  return failure.
INITIALIZE the instance for `api_gpio_0`, the GPIO for the push buttons.
if *the initialization was not successful* **then**
  return failure.
SET the DDR for the LEDs to all outputs.
SET the DDR for the buttons to all inputs.

INITIALIZE the instance for the timer.
if *the initialization was not successful* **then**
  return failure.
ATTACH `TMR_Intr_Handler(void *):void` to handle interrupts on the instance for the timer, bound to the instance for the timer.
SET the compare value of the timer to `0xF8000000`.
SET the timer options to interrupt mode and to reset upon hitting the compare value.

CALL the initialization function `IntcInitFunction(u16, XTmrCtr *, XGpio *):int` using the interrupt controller `xlconcat_0`, the instance for the timer, and the instance for `api_gpio_1`.
if *the initialization was not successful* **then**
  return failure.
START the timer counter.

POLL indefinitely.
return 0 for success.

InterruptSystemSetup

The `InterruptSystemSetup(XScuGic *):int` module function sets enables the button interrupts and sets up the exception handler to the primary interrupt handler. Specifically, it performs Algorithm 4.

Algorithm 4: INTERRUPTSYSTEMSETUP

Input: driver instance data.
Output: `XST_SUCCESS` always.

ENABLE button interrupts on `axi_gpio_0`.
CONFIRM enabling button interrupts with the global enable.

REGISTER the primary interrupt handler, bound to the driver instance data.
ENABLE the exception handling.

---

return success.

IntcInitFunction

The `IntcInitFunction(u16, XTmrCtr *, XGpio *):int` module function does most of the setting up of the interrupt controller `xlconcat_0`. Specifically, it performs Algorithm 5.

Algorithm 5: INTCINITFUNCTION

Input: ID of the device to configure, instance for the timer, instance for a GPIO.
Output: `XSL_SUCCESS` on success; `XSL_FAILURE` if there is an error connecting the GPIO or timer to the handler.

LOOK UP the interrupt controller.
INITIALIZE a driver instance data using the configuration for the interrupt controller.
if *the initialization was not successful* **then**
  return failure.

CALL the initialization function `InterruptSystemSetup(XScuGic *):int` using the interrupt controller driver instance data.
if *the initialization was not successful* **then**
  return failure.

PUT in the driver instance data, a connection from the GPIO interrupt to `BTN_Intr_Handler(void *):void`, bound to the instance of GPIO.
if *the initialization was not successful* **then**
  return failure.

PUT in the driver instance data, a connection from the timer interrupt to `TMR_Intr_Handler(void *):void`, bound to the instance of timer.
if *the initialization was not successful* **then**
  return failure.

ENABLE GPIO interrupts.
CONFIRM enabling GPIO interrupts with the global enable.

ENABLE GPIO interrupts on the the driver instance data.
ENABLE Timer interrupts on the the driver instance data.

---

return success.

Modifications

First, the block design, *zynq_interrupt_system*, was renamed to *zynq_interrupt_system_3_gpios*, using export block design as described in Lab 2.

A new GPIO was added to the block design through the Add IP dialog, and the GPIO `axi_gpio_2` was double clicked to re-customize it. In the Re-customize IP dialog, under board, the GPIO IP Interface was set to the sws 4bits board interface. Then Block automation was ran on both `S_AXI` and `GPIO`. Afterward, the new GPIO was manually accommodated, and Optimize Routing was ran. Then the design was verified. A new HDL Wrapper was created. The new bitstream was generated. Finally, the hardware was exported.

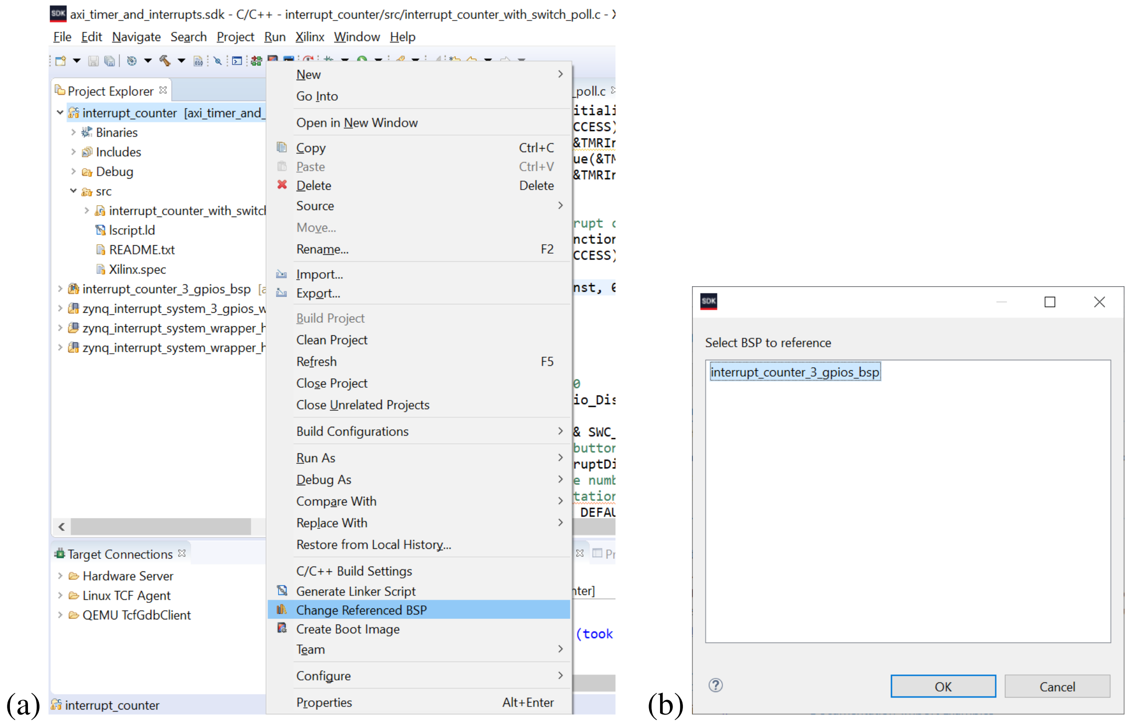

Figure 2: Updating the projects.

Next, the SDK was launched. Since the new GPIO was added, this required an update to the MSS file. The MSS file was updated by creating a new blank Application Project called *interrupt_counter_3_gpios* with the new Hardware Platform *zynq_interrupt_system_3_gpios*. Then interrupt_counter was right clicked, and Change Referenced BSP was selected as in Fig. 2(a). Then *interrupt_counter_3_gpios_bsp* was selected from the dialog, as in Fig. 2(b). As they were no longer needed, *interrupt_counter_**bsp*** and *interrupt_counter_3_gpios*, but not *interrupt_counter_3_gpios_**bsp***, were deleted including their files.

Finally, *interrupt_counter_tut_2B.c* was renamed to *interrupt_counter_with_switch_poll.c* for relevance.

In order to implement the reset switch from step 2 of the manual, the following steps were taken.

First, the number of expiration events before the LED counter increments is now variable. So a global variable `n_expires:static int` was created for that purpose, and `TMR_Intr_Handler(void *data):void` was modified, so that the expiration is compared against `n_expires`.

Additionally, a third GPIO instance was necessary `XGpio SWCInst`, and code for initializing this GPIO and setting its data direction register was added to `main(void):int` along with that of the other peripherals.

Finally, the `while` loop in `main(void):int` was populated with the polling block in

List. 1.

```
// poll SW0
swc_value = XGpio_DiscreteRead(&SWCInst, 1);
// if on
if ((swc_value & SWC_DISABLE_BTNS) == SWC_DISABLE_BTNS) {
    // disable button interrupts
    XGpio_InterruptDisable(&BTNInst, BTN_INT);
    // reset the number of interrupts for LED count
    // incrementation
    n_expires = DEFAULT_N_EXPIRES;
    // reset the LED count display
    led_data = 0b0000;
}
else {
    // re-enable the button interrupts
    XGpio_InterruptEnable(&BTNInst, BTN_INT);
}
```

Listing 1: Polling for switch `SW0`.

To implement step 3, first the `TMR_Intr_Handler(void *):void` function was implemented again to handle resetting and restarting the timer when it is not the case that `tmr_count == n_expires` as seen in List. **??**.

```
void TMR_Intr_Handler(void *data)
{
    if (XTmrCtr_IsExpired(&TMRInst,0)){
        // stop the timer if it expires
        XTmrCtr_Stop(&TMRInst,0);
        // Once timer has expired (n_expires scaled) times,
        // stop, increment counter reset timer and start
        // running again
        if(tmr_count == n_expires * EXPIRATION_SCALE){
            tmr_count = 0;
            led_data++;
            XGpio_DiscreteWrite(&LEDInst, 1, led_data);

        }
        else tmr_count++;
        // in either case, reset and restart the timer
        XTmrCtr_Reset(&TMRInst,0);
        XTmrCtr_Start(&TMRInst,0);
    }
}
```

function.

Listing 2: The fixed `TMR_Intr_Handler(void *):void`

With this implementation, 32 LED counts took $166.62\,\mathrm{s}$, for a rate of $166.62\,\mathrm{s} \div (32\,\mathrm{count} \times \frac{3\,\mathrm{expiration}}{1\,\mathrm{count}}) = 1.7356\,\mathrm{s/expiration}$. Dividing this by 7 gives $0.247\,95\,\mathrm{s/expiration}$, which is about good enough for debouncing.

Then the new `TMR_LOAD` was calculated from the original value of `TMR_LOAD`, which was `0xF8000000`. The delay of the clock is $\texttt{0xFFFFFFFF} - \texttt{0xF8000000} + \texttt{0x1} = \texttt{0x08000000}$. Dividing by 7 gives `0x01249249`. So the new `TMR_LOAD` is $\texttt{0xFFFFFFFF} - \texttt{0x01249249} + \texttt{0x1} = \texttt{0xFEDB6DB7}$.

Additionally, although the delay was scaled down by a factor of 7, the number of expiration events to count to was scaled up by 9 to offset it because this results in 32 LED counts in $171.87\,\mathrm{s}$ with a relative error of 0.031509, more preferable than even 8's $153.37\,\mathrm{s}$ with a relative error of -0.079522.

Conclusions

Appendices