Joseph DeVictoria

Taylor Simons

ECEN 427 Hutchings

September 18, 2014

Real-Time Clock Lab

As we worked through this lab, we came across several issues that we did not anticipate. Each additional feature that we implemented in our code introduced new challenges. By troubleshooting and thinking through each problem as they came, we felt great satisfaction as we overcame the obstacles. We estimate that the entire lab took us roughly four and a half hours to complete. The greatest challenges that we encountered during the course of our lab were fixing some issues that we had with the button debouncing and features.

Although we did not write the skeleton code that was provided for lab, we did gain a firm understanding of how interrupts are enabled and handled in a Xilinx MicroBlaze embedded system. First off, there we had to initialize the GPIO block with a named handle and a device register address for the buttons. The GPIO device also has to have the data direction set to input, its interrupts enabled and its handle registered to a handler function (interrupt\_handle\_dispatcher), using Xilinx macros. This last step allows us to choose which function is called (in the interrupt vector table) when an interrupt is detected. The handler will use pre-defined Xilinx masks for enabling the interrupts. We were then able to write our own handler code to perform specific actions based on which interrupt was sent.

The first thing that the dispatcher does is get the interrupt status and use it to find out which device (IP) sent the interrupt using the XPAR\_FIT\_TIMER\_0\_INTERRUPT\_MASK to identify the fit timer’s interrupts and using XPAR\_PUSH\_BUTTONS\_5BITS\_IP2INTC\_IRPT\_MASK to identify the GPIO block’s interrupts. Depending on which device made the interrupt, an interrupt handler could be called to deal with the interrupt.

We wrote two different interrupt handlers, one for the fit timer, and another for the GPIO device. Our fit timer handler (timer\_interrupt\_handler) basically handled the logic of updating two different timers that we created to help us with the control flow of the clock. One timer was devoted to keeping track of the clock’s time in seconds, and incremented every 100 fit clock cycles (1 second). The other timer allowed us to keep track of debouncing for both the button presses as well as the scrolling feature of the clock. In order to do this we decided that two fit cycles (20 ms) was enough time to debounce the buttons, so as long as the user holds the button down for at least 20 ms, the signal will go through and the appropriate action will be taken (incrementing a value (hours, minutes, seconds) or acknowledging a control button.) . If the debouncing counter showed us that the user held the button down for at least 500ms (.5 s) that we should switch to scrolling mode and automatically update the enabled controls (hour, minute, second) every .5 seconds as long as they kept the button held down. This is actually where the bulk of our time was spent debugging. It was very difficult to get the functionality correct.

The other handler that we wrote (pb\_interrupt\_handler) allowed us to choose what to do when a button is pressed or released. Due to the fact that we did not want any polling to take place, we waited for an interrupt and then pulled in the value of the buttons at that time and reset our debounce timer that we mentioned above. This effectively acted as a flag that would let our other handler implement further logic.

The additional functions that we added were a printing function that split our time in seconds into individual segments for hours, minutes and seconds, and then printed them over the UART in the format specified by the lab. We also added a function that updated each of the time segments based on the combination of buttons pressed per the lab specifications. We added one global flag called updateMode to let us only automatically increment the timer when no time control button was pressed.

Overall it was a satisfying lab, and we were quite successful. We have attached the code that we created for this project to the back of this sheet.

// Taylor Simons & Joseph DeVictoria

// ECEN 427 - FALL 2014.

**#include** "xgpio.h" // Provides access to PB GPIO driver.

**#include** <stdio.h> // xil\_printf and so forth.

**#include** "platform.h" // Enables caching and other system stuff.

**#include** "mb\_interface.h" // provides the microblaze interrupt enables, etc.

**#include** "xintc\_l.h" // Provides handy macros for the interrupt controller.

XGpio gpLED; // This is a handle for the LED GPIO block.

XGpio gpPB; // This is a handle for the push-button GPIO block.

**int** fit\_counter, debouncer, updateMode, scaling;

**int** time, sec, min, hour;

u32 currentButtonState;

//This function adjusts the time based on button presses.

**void** **adjustTime**(**int** cbs){

//the following are masks for all of the button combinations we wish to use

**char** left = cbs&(0x8);

**char** mid = cbs&(0x1);

**char** right = cbs&(0x2);

**char** up = cbs&(0x10);

**char** down = cbs&(0x4);

**char** upDown = cbs&(0x14);

//If no buttons are press then we turn off scaling mode and update mode

**if** (cbs == 0){

updateMode = 0;

scaling = 0;

**return**;

}

//for the right, middle and left buttons we preform logic acordingly

**if** (left){

updateMode = 1;

**if** (up){

**if** (time/3600 >= 23){ //If we go over 23 hours on the display...

time -= 82800; //... we loop back to 00 hours

}

**else**{

time += 3600; //increment the time by an hour

}

}

**if** (down){

**if** (time/3600 <= 0){ //If we go below 00 hours on the display...

time += 82800; //... we loop back to 23 hours

}

**else**{

time -= 3600; //decrement the time by an hour

}

}

**if** (upDown == 0){ //Exit scaling mode

scaling = 0;

}

}

**if** (mid){

updateMode = 1;

**if** (up){

**if** (((time - 3600\*(time/3600))/60) >= 59){ //If we go over 59 minutes on the display...

time -= 3540; //... we loop back to 00 minutes

}

**else**{

time += 60; //increment the time by a minute

}

}

**if** (down){

**if** (((time - 3600\*(time/3600))/60) <= 0){ //If we go below 00 minutes on the display...

time += 3540; //... we loop back to 59 minutes

}

**else**{

time -= 60; //decrement the time by a minute

}

}

**if** (upDown == 0){ //Exit scaling mode

scaling = 0;

}

}

**if** (right){

updateMode = 1;

**if** (up){

**if** ((time % 60) >= 59){ //If we go over 59 seconds on the display...

time -= 59; //... we loop back to 00 seconds

}

**else**{

time++; //increment the time by a second

}

}

**if** (down){

**if** ((time % 60) <= 0){ //If we go below 00 seconds on the display...

time += 59; //... we loop back to 59 seconds

}

**else**{

time--; //decrement the time by a second

}

}

**if** (upDown == 0){

scaling = 0; //Exit scaling mode

}

}

}

**void** **printTime**(){

hour = time / 3600;

min = (time - (3600\*hour)) / 60;

sec = time % 60;

xil\_printf("%02d:%02d:%02d\r", hour, min, sec);

}

// This is invoked in response to a timer interrupt.

// It does 2 things: 1) debounce switches, and 2) advances the time.

**void** **timer\_interrupt\_handler**() {

fit\_counter++;

**if** (scaling == 0){ //If the scaling feature is not set...

**if**(debouncer <= 49){ //...keep counting up to .5 sec, waiting for scaling mode

debouncer++;

}

**if** (debouncer == 2 ){

adjustTime(currentButtonState);//If we have debounced for .02 seconds register a botton press

printTime(); //Print the time after a time change

}

**if** (debouncer == 50 ){ //Set scaling mode after .5 seconds

scaling = 1;

}

}**else** **if** (scaling == 1){ //In scaling mode check the debounce

**if** (debouncer != 50){

debouncer++;

} **else** {

adjustTime(currentButtonState);//If we hit a half second increment the timer

printTime();

debouncer = 0;

}

}

**if** (fit\_counter == 100){ //Update the timer by one second every second

**if** (updateMode == 0){

time++; //Increment the timer

}

printTime(); //Print time after time update

fit\_counter = 0; //reset the fit count

}

}

// This is invoked each time there is a change in the button state (result of a push or a bounce).

**void** **pb\_interrupt\_handler**() {

// Clear the GPIO interrupt.

XGpio\_InterruptGlobalDisable(&gpPB); // Turn off all PB interrupts for now.

// You need to do something here.

currentButtonState = XGpio\_DiscreteRead(&gpPB, 1); // Get the current state of the buttons.

debouncer = 0;

XGpio\_InterruptClear(&gpPB, 0xFFFFFFFF); // Ack the PB interrupt.

XGpio\_InterruptGlobalEnable(&gpPB); // Re-enable PB interrupts.

}

// Main interrupt handler, queries the interrupt controller to see what peripheral

// fired the interrupt and then dispatches the corresponding interrupt handler.

// This routine acks the interrupt at the controller level but the peripheral

// interrupt must be ack'd by the dispatched interrupt handler.

**void** **interrupt\_handler\_dispatcher**(**void**\* ptr) {

**int** intc\_status = XIntc\_GetIntrStatus(XPAR\_INTC\_0\_BASEADDR);

// Check the FIT interrupt first.

**if** (intc\_status & XPAR\_FIT\_TIMER\_0\_INTERRUPT\_MASK){

XIntc\_AckIntr(XPAR\_INTC\_0\_BASEADDR, XPAR\_FIT\_TIMER\_0\_INTERRUPT\_MASK);

timer\_interrupt\_handler();

}

// Check the push buttons.

**if** (intc\_status & XPAR\_PUSH\_BUTTONS\_5BITS\_IP2INTC\_IRPT\_MASK){

XIntc\_AckIntr(XPAR\_INTC\_0\_BASEADDR, XPAR\_PUSH\_BUTTONS\_5BITS\_IP2INTC\_IRPT\_MASK);

pb\_interrupt\_handler();

}

}

**int** **main** (**void**) {

init\_platform();

// Initialize the GPIO peripherals.

**int** success;

// print("hello world\n\r");

success = XGpio\_Initialize(&gpPB, XPAR\_PUSH\_BUTTONS\_5BITS\_DEVICE\_ID);

// success = XGpio\_Initialize(&gpPB, XPAR\_PUSH\_BUTTONS\_5BITS\_DEVICE\_ID);

// Set the push button peripheral to be inputs.

XGpio\_SetDataDirection(&gpPB, 1, 0x0000001F);

// Enable the global GPIO interrupt for push buttons.

XGpio\_InterruptGlobalEnable(&gpPB);

// Enable all interrupts in the push button peripheral.

XGpio\_InterruptEnable(&gpPB, 0xFFFFFFFF);

microblaze\_register\_handler(interrupt\_handler\_dispatcher, NULL);

XIntc\_EnableIntr(XPAR\_INTC\_0\_BASEADDR,

(XPAR\_FIT\_TIMER\_0\_INTERRUPT\_MASK | XPAR\_PUSH\_BUTTONS\_5BITS\_IP2INTC\_IRPT\_MASK));

XIntc\_MasterEnable(XPAR\_INTC\_0\_BASEADDR);

microblaze\_enable\_interrupts();

**while**(1); // Program never ends.

cleanup\_platform();

**return** 0;

}