```c
/*
 * timer.c
 *  Created on: Sep 13, 2016
 *  By Andrew Okazaki and Taylor Cowley
 */
#include "xgpio.h"          // Provides access to PB GPIO driver.
#include <stdio.h>          // xil_printf and so forth.
#include <stdint.h>         // so we don't just use int
#include "platform.h"       // Enables caching and other system stuff.
#include "mb_interface.h"   // provides the microblaze interrupt enables, etc.
#include "xintc_l.h"        // Provides  macros for the interrupt controller.
#include <stdbool.h>

#define BUTTON_UP 0x10          // The mask values for each button
#define BUTTON_DOWN 0x4
#define BUTTON_HOUR 0x8
#define BUTTON_MIN 0x1
#define BUTTON_SEC 0x2
#define NO_BUTTONS 0            // Mask for no buttons
#define BUTTON_DEBOUNCE_TIME 1  // Only need one clock tick to debounce buttons
#define ONE_SECOND           100// 100 ticks in a second
#define ONE_AND_HALF_SECONDS 150// One and a half seconds, in ticks.
#define HALF_SECOND          50 // Half a second, in ticks
#define RESET 0                 // When we reset a timer, it goes to zero
#define PRETTY_NUMBER 10        // For clocks, use 10 to make pretty
#define MAX_SEC 59              // Maximum possible for seconds, mins, and hours
#define MAX_MIN 59
#define MAX_HOUR 23

XGpio gpLED;  // This is a handle for the LED GPIO block.
XGpio gpPB;   // This is a handle for the push-button GPIO block.
int32_t currentButtonState = RESET; // Start with no buttons being pushed
int32_t timerCount = RESET;         // Seconds timer not running
int32_t debounce_timer_up = RESET;  // Timers for debouncing the up and
int32_t debounce_timer_down = RESET;// down buttons

int32_t hours = PRETTY_NUMBER;      // Variables to store the time
int32_t minutes = PRETTY_NUMBER;    // Start the time at
int32_t seconds = PRETTY_NUMBER;    // nice friendly numbers

// We call this once a second to update the time
void evaluate();

// 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() {
    if(currentButtonState == NO_BUTTONS){   // Only tick if no pushed buttons
        timerCount++;
        if(timerCount >= ONE_SECOND){       // Wait a second
            timerCount = RESET;             // Reset the timer
            seconds++;                      // Tick seconds
            evaluate();                     // Fix the time
        }
    }else{
        // The hour button is being pushed
        if(currentButtonState & BUTTON_HOUR){
            // The up button is being pushed
            if(currentButtonState & BUTTON_UP){
```

```
59                  debounce_timer_up++;        // increase time held
60                  if(debounce_timer_up == BUTTON_DEBOUNCE_TIME){      // READY
61                      hours++;                // increase hours
62                      evaluate();             // and fix time
63                  // For every half second over one and a half seconds
64                  }else if(debounce_timer_up % HALF_SECOND == 0
65                          && debounce_timer_up > ONE_AND_HALF_SECONDS){
66                      hours++;                // we also increase
67                      evaluate();             // and fix the time
68                  }
69              }else{debounce_timer_up =RESET;}// Up is not pushed, reset debounce
70              //The down button is being pushed
71              if(currentButtonState & BUTTON_DOWN){
72                  debounce_timer_down++;      // Increase the time held
73                  if(debounce_timer_down == BUTTON_DEBOUNCE_TIME){    // READY
74                      hours--;                // decrease hours
75                      evaluate();             // And fix time
76                  // For every half second over one and a half seconds
77                  }else if(debounce_timer_down % HALF_SECOND == 0
78                          && debounce_timer_down > ONE_AND_HALF_SECONDS){
79                      hours--;                // decrease and fix time
80                      evaluate();
81                  }
82              }else{debounce_timer_down=RESET;}// Not being held, reset held timer
83          }
84          // The minute button is being pushed
85          if(currentButtonState & BUTTON_MIN){
86              // The up button is being pushed
87              if(currentButtonState & BUTTON_UP){
88                  debounce_timer_up++;        // Increase held time
89                  if(debounce_timer_up == BUTTON_DEBOUNCE_TIME){      // READY
90                      minutes++;              // increase minutes and fix
91                      evaluate();
92                  // For every half second over one and a half seconds
93                  }else if(debounce_timer_up % HALF_SECOND == 0
94                          && debounce_timer_up > ONE_AND_HALF_SECONDS){
95                      minutes++;              // Also tick and fix
96                      evaluate();
97                  }
98              }else{debounce_timer_up =RESET;}// Not being pushed; reset timer
99              // The down button is being pushed
100             if(currentButtonState & BUTTON_DOWN){
101                 debounce_timer_down++;      // Increase time pressed
102                 if(debounce_timer_down == BUTTON_DEBOUNCE_TIME){    // READY
103                     minutes--;              // decrease minutes
104                     evaluate();
105                 // For every half second over one and a half seconds
106                 }else if(debounce_timer_down % HALF_SECOND == 0
107                         && debounce_timer_down > ONE_AND_HALF_SECONDS){
108                     minutes--;              // Tick and fix
109                     evaluate();
110                 }
111             }else{debounce_timer_down=RESET;}// Not being held, reset timer
112         }
113         // The second button is being pushed
114         if(currentButtonState & BUTTON_SEC){
115             // The up button is being pushed
116             if(currentButtonState & BUTTON_UP){
```

```
117                 debounce_timer_up++;        // Increase held timer
118                 if(debounce_timer_up == BUTTON_DEBOUNCE_TIME){      // READY
119                     seconds++;              // Tick and fix
120                     evaluate();
121                 // For every half second over one and a half seconds
122                 }else if(debounce_timer_up % HALF_SECOND == 0
123                         && debounce_timer_up > ONE_AND_HALF_SECONDS){
124                     seconds++;              // Tick and fix
125                     evaluate();
126                 }
127             }else{debounce_timer_up =RESET;}// Not being held, reset timer
128             // The down button is being pushed
129             if(currentButtonState & BUTTON_DOWN){
130                 debounce_timer_down++;      // Increase held timer
131                 if(debounce_timer_down == BUTTON_DEBOUNCE_TIME){    // READY
132                     seconds--;              // tick and fix
133                     evaluate();
134                 // For every half second over one and a half seconds
135                 }else if(debounce_timer_down % HALF_SECOND == 0
136                         && debounce_timer_down > ONE_AND_HALF_SECONDS){
137                     seconds--;              // tick and fix
138                     evaluate();
139                 }
140             }else{debounce_timer_down=RESET;}// Not being held, reset held timer
141         }
142     }
143 }
144
145 // This updates our time variables to make time sense
146 // This is also what displays the time
147 void evaluate(){
148     // These if statements make the time go up
149     if(seconds > MAX_SEC){  // Seconds are between 0 and 59
150         seconds = RESET;
151         minutes++;  // new minute!
152     }
153     if(minutes > MAX_MIN){      // Minutes are between 0 and 59
154         minutes = RESET;
155         hours++;    // new hour!
156     }
157     if(hours > MAX_HOUR){       // Hours are between 0 and 23
158         hours = RESET;
159     }
160
161     // These if statements make the time go down
162     if(seconds < 0){        // Can't have negative seconds
163         seconds = MAX_SEC;
164         minutes--;  // Subtract a minute
165     }
166     if(minutes < 0){        // Can't have negative minutes
167         minutes = MAX_MIN;
168         hours--;    // Subtract an hour
169     }
170     if(hours < 0){      // Can't have negative hours
171         hours = MAX_HOUR;
172     }
173     // Prints the time. We only use a carriage return so we can overwrite it
174     xil_printf("\e[104m %02d:%02d:%02d \e[49m\r", hours, minutes, seconds);
```

```
175 }
176
177 // This is invoked each time there is a change in the button state
178 // (result of a push or a bounce).
179 void pb_interrupt_handler() {
180     // Clear the GPIO interrupt.
181     XGpio_InterruptGlobalDisable(&gpPB);                // Off PB interrupts now
182     currentButtonState = XGpio_DiscreteRead(&gpPB, 1);  // Get state of buttons.
183     // This was all that was necessary. Just update the button state
184     XGpio_InterruptClear(&gpPB, 0xFFFFFFFF);            // Ack the PB interrupt.
185     XGpio_InterruptGlobalEnable(&gpPB);                 // Enable PB interrupts.
186 }
187
188 // Main interrupt handler, queries interrupt controller to see what peripheral
189 // fired the interrupt and then dispatches the corresponding interrupt handler.
190 // This routine acks the interrupt at the controller level but the peripheral
191 // interrupt must be ack'd by the dispatched interrupt handler.
192 // Question: Why is timer_interrupt_handler() called after ack'ing controller
193 // but pb_interrupt_handler() is called before ack'ing the interrupt controller?
194 void interrupt_handler_dispatcher(void* ptr) {
195     int intc_status = XIntc_GetIntrStatus(XPAR_INTC_0_BASEADDR);
196     // Check the FIT interrupt first.
197     if (intc_status & XPAR_FIT_TIMER_0_INTERRUPT_MASK){
198         XIntc_AckIntr(XPAR_INTC_0_BASEADDR, XPAR_FIT_TIMER_0_INTERRUPT_MASK);
199         timer_interrupt_handler();  // It was a timer interrupt! call that fn
200     }
201     // Check the push buttons.
202     if (intc_status & XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK){
203         pb_interrupt_handler();     // It was a button interrupt!
204         XIntc_AckIntr(XPAR_INTC_0_BASEADDR, // Acknowledge the interrupt
205                 XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK);
206     }
207 }
208
209 int main (void) {
210     init_platform();
211     // Initialize the GPIO peripherals.
212     int32_t success;
213     print("\n\rHello . Let's have a fun \e[31m\e[1mtime \e[21m\e[0m\n\r");
214     success = XGpio_Initialize(&gpPB, XPAR_PUSH_BUTTONS_5BITS_DEVICE_ID);
215     // Set the push button peripheral to be inputs.
216     XGpio_SetDataDirection(&gpPB, 1, 0x0000001F);
217     // Enable the global GPIO interrupt for push buttons.
218     XGpio_InterruptGlobalEnable(&gpPB);
219     // Enable all interrupts in the push button peripheral.
220     XGpio_InterruptEnable(&gpPB, 0xFFFFFFFF);
221     // Register the interrupt handler
222     microblaze_register_handler(interrupt_handler_dispatcher, NULL);
223     // And enable interrupts
224     XIntc_EnableIntr(XPAR_INTC_0_BASEADDR,
225             (XPAR_FIT_TIMER_0_INTERRUPT_MASK |
226                     XPAR_PUSH_BUTTONS_5BITS_IP2INTC_IRPT_MASK));
227     // Master the enable
228     XIntc_MasterEnable(XPAR_INTC_0_BASEADDR);
229     // And enable again
230     microblaze_enable_interrupts();
231
232     while(1);  // Program never ends.
```

```
233     cleanup_platform();
234     return 0;
235 }
236
```