Lab 04 Report: Vivado AXI Timer and Interrupts

Peter Mowen
peter.mowen@temple.edu

Summary

The understanding of interrupts will be expanded to include the AXI timer as a source. Interrupt multiplexing and configuring a GPIO IP to handle two channels will be introduced. A finite state machine will be used to turn the switches into a combination lock.

Introduction

The main objective of this lab is to introduce timers on the Zybo board. Exercise 2D from *The Zynq Book Tutorial* will be used as a basis for this lab. Furthermore, this lab extends interrupts to include timers and GPIO IPs to include multiple devices on a single GPIO IP. The design specification for this lab is summarized below:

1. Exercise 2D from *The Zynq Book Tutorial* has the Zybo board updating the LEDs based on the timer. Moreover, the buttons add their value to the led count when pressed.
2. The switches will be used as a combination lock to lock the buttons and send the LEDs into a locked state based on the following sequences:
   a. To lock the board, the switches must be thrown in the following order: SW3, SW1, SW2, SW0. If at any point during the locking, an incorrect switch is thrown, then all the switches must be turned OFF before the locking sequence can begin again.
   b. To unlock the board, the switches must be thrown in the following order: SW3, SW1, SW2, SW0. If at any point during the unlocking, an incorrect switch is thrown, then all the switches must be turned ON before the locking sequence can begin again.
   c. Once in the locked state, the buttons are to be deactivated, and the LEDs are to alternate between 1010 and 0101.
   d. When unlocked, the LEDs and buttons are to behave as in exercise 2D.

A finite state machine will be designed to meet the above specifications. Based on the its current state and the switch input, it will move from unlocked to locked and back.

Discussion

Hardware Setup in Vivado HLx

The following is a screenshot of the final hardware design used for this lab:

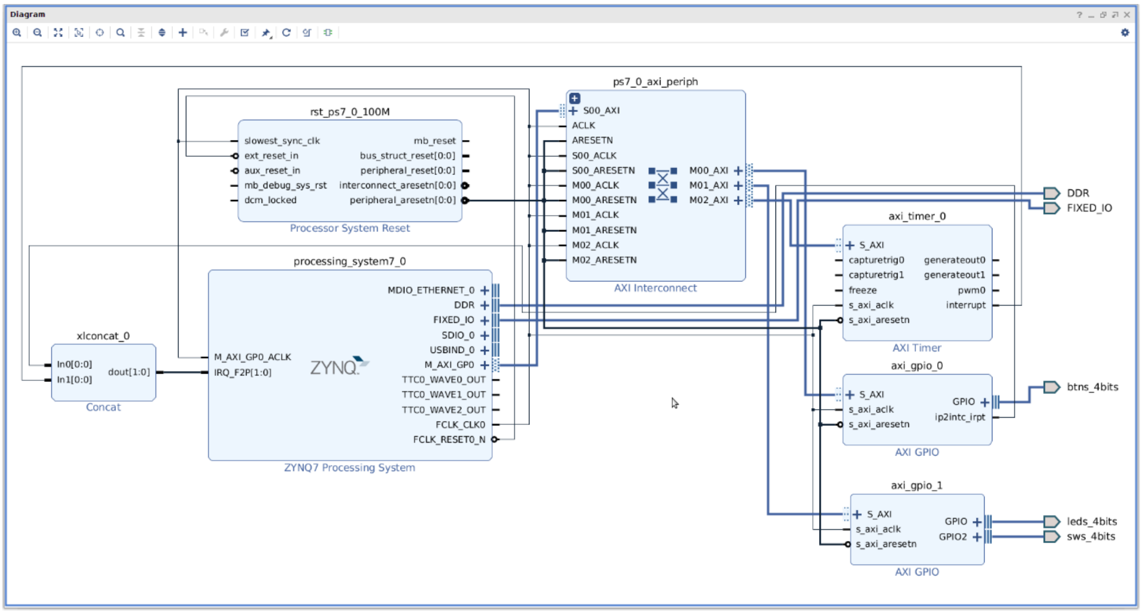


Figure 1: Hardware Block Diagram from Vivado HLx

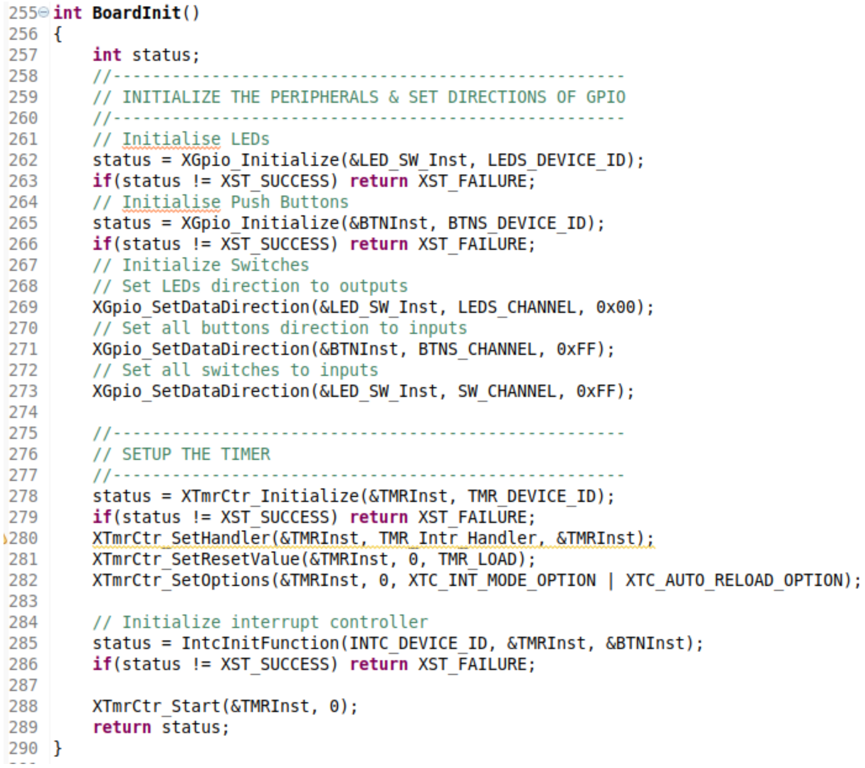
This block diagram is heavily based on Exercise 2D from *The Zynq Book Tutorials* (Louise H. Crocket, 2015). This design adds the AXI Timer IP to the block diagram from Lab 03. Both the timer and the buttons require interrupts. They are both connected to the Concat IP, which acts as a multiplexer for the interrupt line.

Another addition is in GPIO 1. This GPIO has two channels. The first channel is used for the LEDs and the second channel is used for the switches. The switches were added to this IP instead of GPIO 0 because they are not supposed to trigger an interrupt.

Analysis of Exercise 2D Code

Much of the code used in the lab was also used in Lab 03. That code will be reviewed here, and the new code will be introduced.

The following is a screenshot of the `BoardInit()` function created by this student. It takes the board initialization functions from exercise 2D and encapsulates them in their own function.



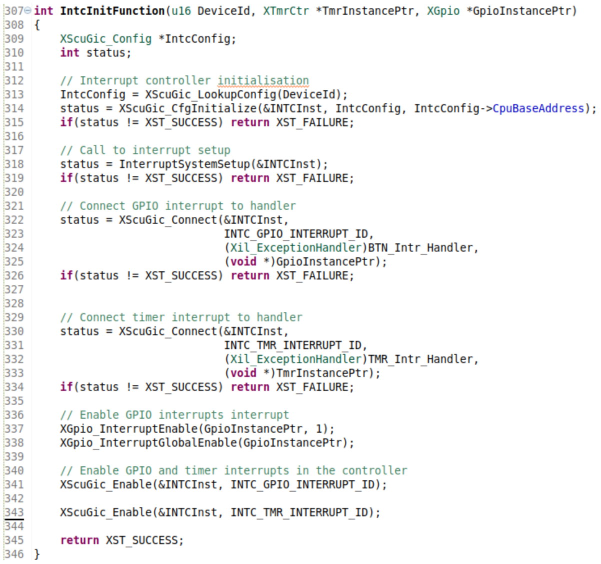
```
255 int BoardInit()
256 {
257     int status;
258     //--------------------------------------------------
259     // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
260     //--------------------------------------------------
261     // Initialise LEDs
262     status = XGpio_Initialize(&LED_SW_Inst, LEDS_DEVICE_ID);
263     if(status != XST_SUCCESS) return XST_FAILURE;
264     // Initialise Push Buttons
265     status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
266     if(status != XST_SUCCESS) return XST_FAILURE;
267     // Initialize Switches
268     // Set LEDs direction to outputs
269     XGpio_SetDataDirection(&LED_SW_Inst, LEDS_CHANNEL, 0x00);
270     // Set all buttons direction to inputs
271     XGpio_SetDataDirection(&BTNInst, BTNS_CHANNEL, 0xFF);
272     // Set all switches to inputs
273     XGpio_SetDataDirection(&LED_SW_Inst, SW_CHANNEL, 0xFF);
274
275     //--------------------------------------------------
276     // SETUP THE TIMER
277     //--------------------------------------------------
278     status = XTmrCtr_Initialize(&TMRInst, TMR_DEVICE_ID);
279     if(status != XST_SUCCESS) return XST_FAILURE;
280     XTmrCtr_SetHandler(&TMRInst, TMR_Intr_Handler, &TMRInst);
281     XTmrCtr_SetResetValue(&TMRInst, 0, TMR_LOAD);
282     XTmrCtr_SetOptions(&TMRInst, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
283
284     // Initialize interrupt controller
285     status = IntcInitFunction(INTC_DEVICE_ID, &TMRInst, &BTNInst);
286     if(status != XST_SUCCESS) return XST_FAILURE;
287
288     XTmrCtr_Start(&TMRInst, 0);
289     return status;
290 }
```

Figure 2: Code for Board Initialization Function

Lines 262 through 273 initialize the LEDs, buttons, and switches, then sets the data direction for each. Note that a single object, `LED_SW_Inst`, is used for both the LEDs and switches. As mentioned previously, both the LEDs and switches are on the same AXI GPIO, each on their own channel. Their channels are defined in the preprocessor definitions.

Lines 278 through 288 setup the timer. Line 278 calls the `XTmrCtr_Initialize` function. This function takes two inputs. The first is the address of the timer instance and the second is the timer device ID. This function is analogous to the GPIO initialization function. Line 280 ties the timer instance to the timer handler. Line 281 sets the initial value, `TMR_LOAD`, for the timer. The timer will assume this value every time it is reset. Line 282 sets the options for the timer. The last argument ORs the mask for interrupt mode and auto-reload mode. This allows the timer to call interrupts and auto-reload. Line 285 passes the addresses of the timer and button instances to the interrupt controller initialization function. Finally, line 288 starts the timer.

The following is a screenshot of the `IntcInitFunction()` taken from exercise 2D:

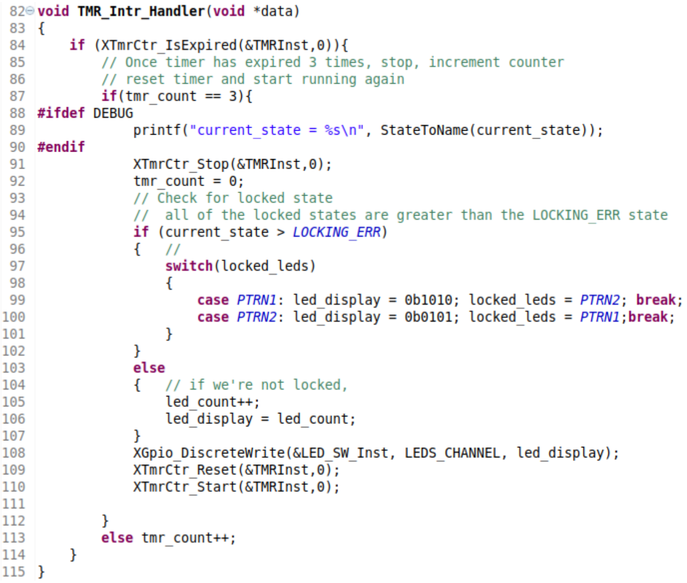


```
307 int IntcInitFunction(u16 DeviceId, XTmrCtr *TmrInstancePtr, XGpio *GpioInstancePtr)
308 {
309     XScuGic_Config *IntcConfig;
310     int status;
311 
312     // Interrupt controller initialisation
313     IntcConfig = XScuGic_LookupConfig(DeviceId);
314     status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
315     if(status != XST_SUCCESS) return XST_FAILURE;
316 
317     // Call to interrupt setup
318     status = InterruptSystemSetup(&INTCInst);
319     if(status != XST_SUCCESS) return XST_FAILURE;
320 
321     // Connect GPIO interrupt to handler
322     status = XScuGic_Connect(&INTCInst,
323                              INTC_GPIO_INTERRUPT_ID,
324                              (Xil_ExceptionHandler)BTN_Intr_Handler,
325                              (void *)GpioInstancePtr);
326     if(status != XST_SUCCESS) return XST_FAILURE;
327 
328 
329     // Connect timer interrupt to handler
330     status = XScuGic_Connect(&INTCInst,
331                              INTC_TMR_INTERRUPT_ID,
332                              (Xil_ExceptionHandler)TMR_Intr_Handler,
333                              (void *)TmrInstancePtr);
334     if(status != XST_SUCCESS) return XST_FAILURE;
335 
336     // Enable GPIO interrupts interrupt
337     XGpio_InterruptEnable(GpioInstancePtr, 1);
338     XGpio_InterruptGlobalEnable(GpioInstancePtr);
339 
340     // Enable GPIO and timer interrupts in the controller
341     XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
342 
343     XScuGic_Enable(&INTCInst, INTC_TMR_INTERRUPT_ID);
344 
345     return XST_SUCCESS;
346 }
```

Figure 3: Interrupt Controller Initialization Function

Lines 313 through 326 are exactly the code from Lab 03 and were discussed in detail there. Lines 330 through 334 connect the timer to the interrupt controller. The `InterruptSystemSetup()` function was also discussed in the Lab 03 report and will not be discussed again here. The first argument to `XScuGic_Connect()` is the address of the interrupt controller instance. The next argument is the interrupt ID for the timer. The third argument is the handle to the interrupt service routine which the timer interrupt will call. The final argument is the address of the timer instance, which is passed into this function from the `BoardInit()` function. Lines 337 through 341 enable the interrupt for the GPIO and line 343 enables the timer interrupt.

The following screenshot shows the code for the timer ISR:



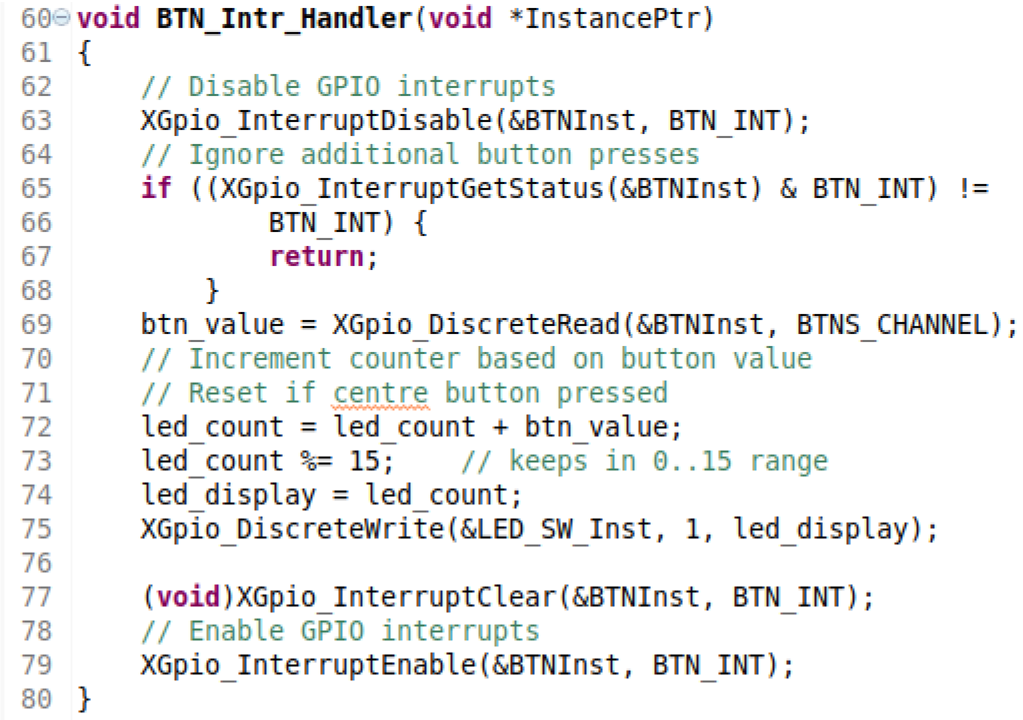
```
void TMR_Intr_Handler(void *data)
{
    if (XTmrCtr_IsExpired(&TMRInst,0)){
        // Once timer has expired 3 times, stop, increment counter
        // reset timer and start running again
        if(tmr_count == 3){
#ifdef DEBUG
            printf("current_state = %s\n", StateToName(current_state));
#endif
            XTmrCtr_Stop(&TMRInst,0);
            tmr_count = 0;
            // Check for locked state
            //  all of the locked states are greater than the LOCKING_ERR state
            if (current_state > LOCKING_ERR)
            {   //
                switch(locked_leds)
                {
                    case PTRN1: led_display = 0b1010; locked_leds = PTRN2; break;
                    case PTRN2: led_display = 0b0101; locked_leds = PTRN1;break;
                }
            }
            else
            {   // if we're not locked,
                led_count++;
                led_display = led_count;
            }
            XGpio_DiscreteWrite(&LED_SW_Inst, LEDS_CHANNEL, led_display);
            XTmrCtr_Reset(&TMRInst,0);
            XTmrCtr_Start(&TMRInst,0);

        }
        else tmr_count++;
    }
}
```

Figure 4: Timer Interrupt Service Routine

Once the program enters the timer ISR, the first thing it does is check whether the timer has expired. If it has, it checks to see if the timer count is three. If it is three, then the code from lines 88 through 110 run. In the original exercise, the only thing that happened is this block was that the led count was incremented, then written to the LEDs. However, the code above changes what is to be displayed on the LEDs based on the current state of the finite state machine and the `locked_leds` state variable. Once the LEDs have been updated, the timer instance is reset, then started again for the next run.

The following is a screenshot of the code for the button ISR:



```
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;
        }
    btn_value = XGpio_DiscreteRead(&BTNInst, BTNS_CHANNEL);
    // Increment counter based on button value
    // Reset if centre button pressed
    led_count = led_count + btn_value;
    led_count %= 15;    // keeps in 0..15 range
    led_display = led_count;
    XGpio_DiscreteWrite(&LED_SW_Inst, 1, led_display);

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

Figure 5: Button Interrupt Service Routine

This code is almost identical to the code in Lab 03. First, the GPIO interrupt is disabled. Then the interrupt status of the button instance is checked by ANDing it with the button interrupt mask. If this result is not equal to the button interrupt mask, then the ISR is exited. Else, the button value is read in and added to the LED count. The led count is then set to the led count modulo 15. This keeps its value between zero and fifteen. Next the LED display is set to the LED count and is then written to the LEDs. Finally, the interrupt flag for the button instance is cleared and the GPIO interrupts are reenabled.

Control Logic

The following finite state machine was created to define how to move between states to meet the specification outlined in the Introduction:

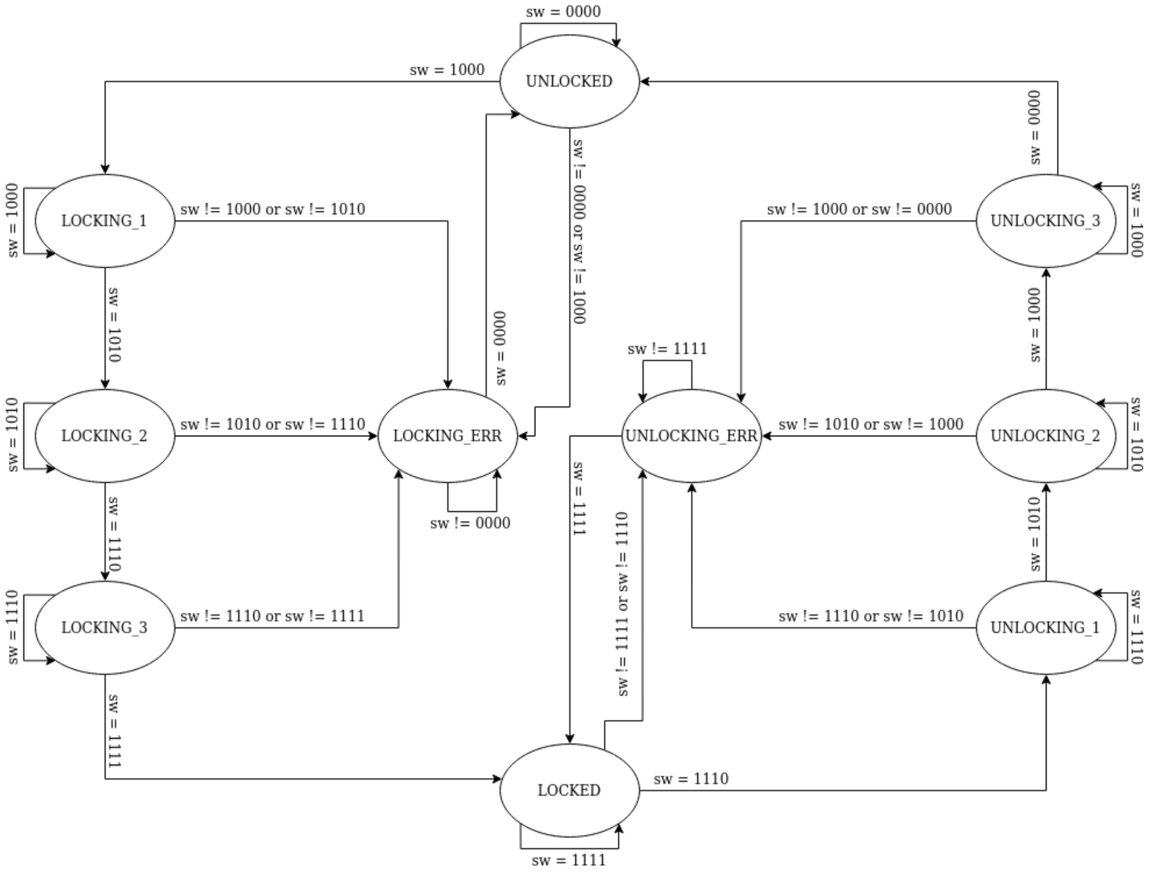
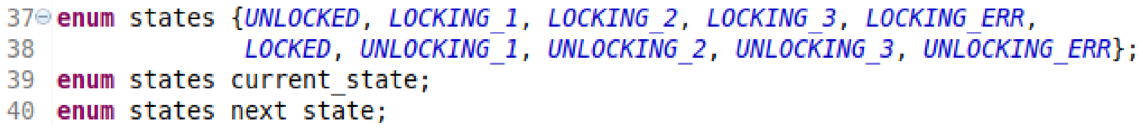


Figure 6: Control Logic Finite State Machine

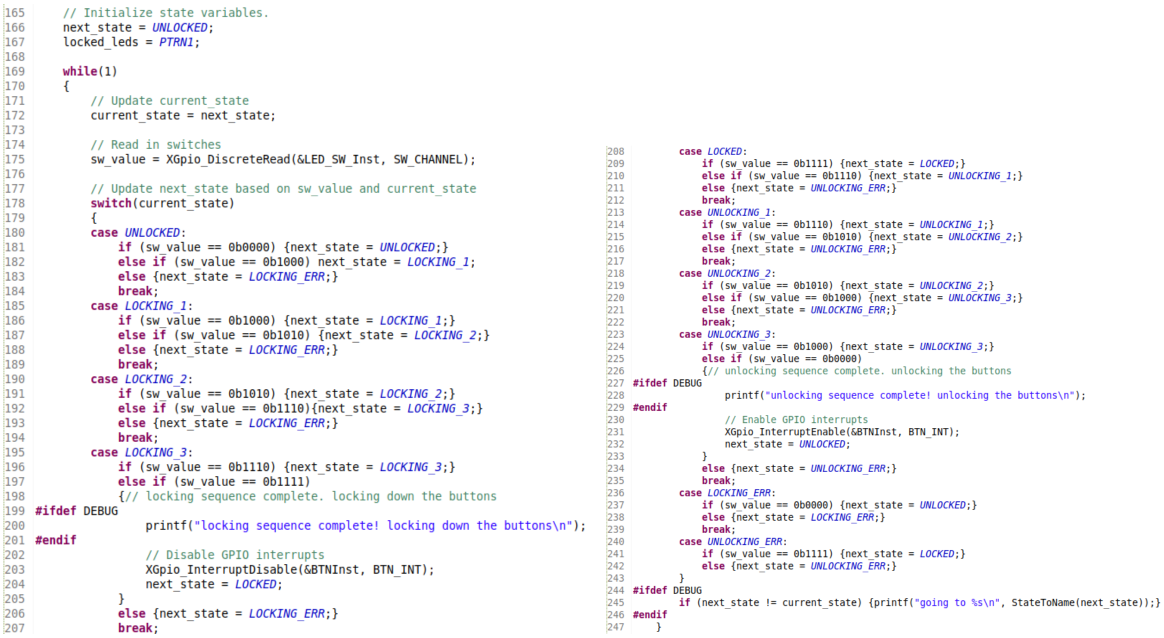
The states were defined using the enum variable time as shown in the screenshot below:



```
enum states {UNLOCKED, LOCKING_1, LOCKING_2, LOCKING_3, LOCKING_ERR,
             LOCKED, UNLOCKING_1, UNLOCKING_2, UNLOCKING_3, UNLOCKING_ERR};
enum states current_state;
enum states next_state;
```

Figure 7: State variable declaration

Similar to previous labs, the `current_state` and `next_state` variables were created and used to control the flow of the FSM. The following screenshots show the code which implements the finite state machine:



```c
    // Initialize state variables.
    next_state = UNLOCKED;
    locked_leds = PTRN1;

    while(1)
    {
        // Update current_state
        current_state = next_state;

        // Read in switches
        sw_value = XGpio_DiscreteRead(&LED_SW_Inst, SW_CHANNEL);

        // Update next_state based on sw_value and current_state
        switch(current_state)
        {
        case UNLOCKED:
            if (sw_value == 0b0000) {next_state = UNLOCKED;}
            else if (sw_value == 0b1000) next_state = LOCKING_1;
            else {next_state = LOCKING_ERR;}
            break;
        case LOCKING_1:
            if (sw_value == 0b1000) {next_state = LOCKING_1;}
            else if (sw_value == 0b1010) {next_state = LOCKING_2;}
            else {next_state = LOCKING_ERR;}
            break;
        case LOCKING_2:
            if (sw_value == 0b1010) {next_state = LOCKING_2;}
            else if (sw_value == 0b1110){next_state = LOCKING_3;}
            else {next_state = LOCKING_ERR;}
            break;
        case LOCKING_3:
            if (sw_value == 0b1110) {next_state = LOCKING_3;}
            else if (sw_value == 0b1111)
            {// locking sequence complete. locking down the buttons
#ifdef DEBUG
                printf("locking sequence complete! locking down the buttons\n");
#endif
                // Disable GPIO interrupts
                XGpio_InterruptDisable(&BTNInst, BTN_INT);
                next_state = LOCKED;
            }
            else {next_state = LOCKING_ERR;}
            break;
        case LOCKED:
            if (sw_value == 0b1111) {next_state = LOCKED;}
            else if (sw_value == 0b1110) {next_state = UNLOCKING_1;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_1:
            if (sw_value == 0b1110) {next_state = UNLOCKING_1;}
            else if (sw_value == 0b1010) {next_state = UNLOCKING_2;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_2:
            if (sw_value == 0b1010) {next_state = UNLOCKING_2;}
            else if (sw_value == 0b1000) {next_state = UNLOCKING_3;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_3:
            if (sw_value == 0b1000) {next_state = UNLOCKING_3;}
            else if (sw_value == 0b0000)
            {// unlocking sequence complete. unlocking the buttons
#ifdef DEBUG
                printf("unlocking sequence complete! unlocking the buttons\n");
#endif
                // Enable GPIO interrupts
                XGpio_InterruptEnable(&BTNInst, BTN_INT);
                next_state = UNLOCKED;
            }
            else {next_state = UNLOCKING_ERR;}
            break;
        case LOCKING_ERR:
            if (sw_value == 0b0000) {next_state = UNLOCKED;}
            else {next_state = LOCKING_ERR;}
            break;
        case UNLOCKING_ERR:
            if (sw_value == 0b1111) {next_state = LOCKED;}
            else {next_state = UNLOCKING_ERR;}
        }
#ifdef DEBUG
        if (next_state != current_state) {printf("going to %s\n", StateToName(next_state));}
#endif
    }
```

Figure 8: Finite State Machine Code

The above code is the meat of the `main` function. First, the `next_state` and `locked_leds` are initialized. The `locked_leds` variable was seen in the timer ISR and is used to determine which pattern is to be displayed on the LEDs. After entering the infinite while loop, the `current_state` is updated to the `next_state`, then the switches are read into the `sw_value` variable. The `next_state` variable is then updated based on the `current_state` and the `sw_value` variables. The only way from `UNLOCKED` to `LOCKED` and back is by switching the buttons in the correct order. If a switch is flipped out of sequence, the FSM goes into one of the error states, `LOCKING_ERR` or `UNLOCKING_ERR`. The only way out of these states is to flip all the switches off or on respectively.

Results

The picture below shows all the switches down and an LED display that is only possible while the board is unlocked.

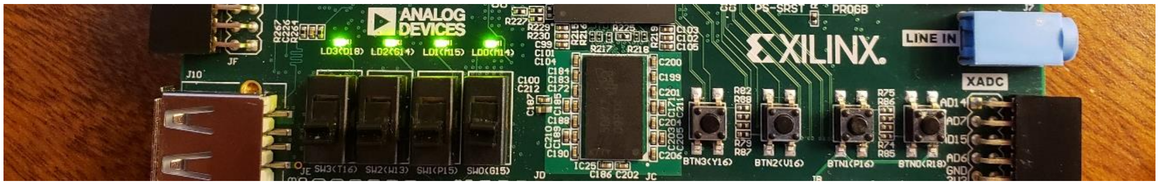


Figure 9: Unlocked board

The picture below shows all the switches up and an LED displaying that is one of the locked patterns:

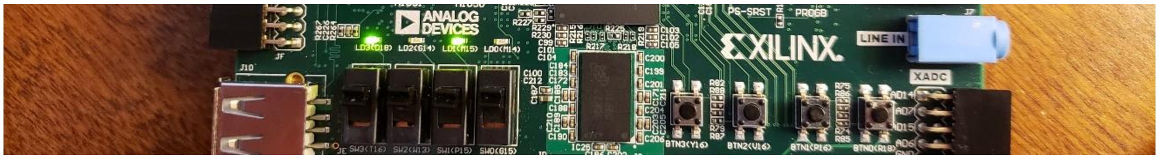


Figure 10: Locked board

The following link is to a YouTube video showing a full demonstration:

https://youtu.be/tuuWLyO_gNM

Conclusions

This lab successfully introduced timers on the Zybo board. A timer was used to update the LEDs based on the current state and the led count. This lab further solidified the importance of finite state machines in standalone OS embedded system design, as this was the third lab that could be solved by creating one. Interrupts and GPIOs on Zybo were expanded upon successfully. Interrupts were expanded to include timers and GPIOs were expanded to include having two devices on the same GPIO IP.

Appendix

C Code

```
#include "xparameters.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xil_printf.h"

// Parameter definitions
#define INTC_DEVICE_ID                  XPAR_PS7_SCUGIC_0_DEVICE_ID
#define TMR_DEVICE_ID                   XPAR_TMRCTR_0_DEVICE_ID
#define BTNS_DEVICE_ID                  XPAR_AXI_GPIO_0_DEVICE_ID
#define BTNS_CHANNEL                    1
#define LEDS_DEVICE_ID                  XPAR_AXI_GPIO_1_DEVICE_ID
#define LEDS_CHANNEL                    1
#define SW_DEVICE_ID                    XPAR_AXI_GPIO_1_DEVICE_ID
#define SW_CHANNEL                      2

#define INTC_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define INTC_TMR_INTERRUPT_ID XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR

#define BTN_INT                         XGPIO_IR_CH1_MASK
#define TMR_LOAD                        0xF8000000

#define printf xil_printf

#define DEBUG

XGpio LED_SW_Inst, BTNInst;
XScuGic INTCInst;
XTmrCtr TMRInst;
static int led_count;
static int led_display;
static int btn_value;
static int sw_value;
static int tmr_count;

enum states {UNLOCKED, LOCKING_1, LOCKING_2, LOCKING_3, LOCKING_ERR,
             LOCKED, UNLOCKING_1, UNLOCKING_2, UNLOCKING_3, UNLOCKING_ERR};
enum states current_state;
enum states next_state;
enum event_state {PTRN1, PTRN2};
enum event_state locked_leds;

//----------------------------------------------------
// PROTOTYPE FUNCTIONS
//----------------------------------------------------
static void BTN_Intr_Handler(void *baseaddr_p);
static void TMR_Intr_Handler(void *baseaddr_p);
static int InterruptSystemSetup(XScuGic *XScuGicInstancePtr);
static int IntcInitFunction(u16 DeviceId, XTmrCtr *TmrInstancePtr, XGpio *GpioInstancePtr);
static int BoardInit();
static char* StateToName(int state);

//----------------------------------------------------
// INTERRUPT HANDLER FUNCTIONS
// - called by the timer, button interrupt, performs
// - LED flashing
//----------------------------------------------------
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;
        }
    btn_value = XGpio_DiscreteRead(&BTNInst, BTNS_CHANNEL);
    // Increment counter based on button value
    // Reset if centre button pressed
    led_count = led_count + btn_value;
    led_count %= 15;    // keeps in 0..15 range
    led_display = led_count;
  XGpio_DiscreteWrite(&LED_SW_Inst, 1, led_display);

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

void TMR_Intr_Handler(void *data)
{
    if (XTmrCtr_IsExpired(&TMRInst,0)){
        // Once timer has expired 3 times, stop, increment counter
        // reset timer and start running again
        if(tmr_count == 3){
#ifdef DEBUG
            printf("current_state = %s\n", StateToName(current_state));
#endif
            XTmrCtr_Stop(&TMRInst,0);
            tmr_count = 0;
            // Check for locked state
            //    all of the locked states are greater than the LOCKING_ERR state
            if (current_state > LOCKING_ERR)
            {   //
                switch(locked_leds)
                {
                    case PTRN1: led_display = 0b1010; locked_leds = PTRN2; break;
                    case PTRN2: led_display = 0b0101; locked_leds = PTRN1;break;
                }
            }
            else
            {   // if we're not locked,
                led_count++;
                led_display = led_count;
            }
            XGpio_DiscreteWrite(&LED_SW_Inst, LEDS_CHANNEL, led_display);
            XTmrCtr_Reset(&TMRInst,0);
            XTmrCtr_Start(&TMRInst,0);

        }
        else tmr_count++;
    }
}

// Function to change state variable number to corresponding string name
char* StateToName(int state)
```

```
{
        char* state_name;
        switch (state)
        {
                case UNLOCKED:
                        state_name = "UNLOCKED";
                        break;
                case LOCKING_1:
                        state_name = "LOCKING_1";
                        break;
                case LOCKING_2:
                        state_name = "LOCKING_2";
                        break;
                case LOCKING_3:
                        state_name = "LOCKING_3";
                        break;
                case LOCKING_ERR:
                        state_name = "LOCKING_ERR";
                        break;
                case LOCKED:
                        state_name = "LOCKED";
                        break;
                case UNLOCKING_1:
                        state_name = "UNLOCKING_1";
                        break;
                case UNLOCKING_2:
                        state_name = "UNLOCKING_2";
                        break;
                case UNLOCKING_3:
                        state_name = "UNLOCKING_3";
                        break;
                case UNLOCKING_ERR:
                        state_name = "UNLOCKING_ERR";
                        break;
        }
        return state_name;
}

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

        next_state = UNLOCKED;
        locked_leds = PTRN1;

        while(1)
        {
                // Update current_state
                current_state = next_state;

                // Read in switches
                sw_value = XGpio_DiscreteRead(&LED_SW_Inst, SW_CHANNEL);

                // Update next_state based on sw_value and current_state
                switch(current_state)
                {
                case UNLOCKED:
                        if (sw_value == 0b0000) {next_state = UNLOCKED;}
                        else if (sw_value == 0b1000) next_state = LOCKING_1;
                        else {next_state = LOCKING_ERR;}
                        break;
                case LOCKING_1:
                        if (sw_value == 0b1000) {next_state = LOCKING_1;}
                        else if (sw_value == 0b1010) {next_state = LOCKING_2;}
                        else {next_state = LOCKING_ERR;}
                        break;
                case LOCKING_2:
                        if (sw_value == 0b1010) {next_state = LOCKING_2;}
                        else if (sw_value == 0b1110){next_state = LOCKING_3;}
                        else {next_state = LOCKING_ERR;}
                        break;
                case LOCKING_3:
                        if (sw_value == 0b1110) {next_state = LOCKING_3;}
                        else if (sw_value == 0b1111)
```

```
            {// locking sequence complete. locking down the buttons
#ifdef DEBUG
                printf("locking sequence complete! locking down the buttons\n");
#endif
                // Disable GPIO interrupts
                XGpio_InterruptDisable(&BTNInst, BTN_INT);
                next_state = LOCKED;
            }
            else {next_state = LOCKING_ERR;}
            break;
        case LOCKED:
            if (sw_value == 0b1111) {next_state = LOCKED;}
            else if (sw_value == 0b1110) {next_state = UNLOCKING_1;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_1:
            if (sw_value == 0b1110) {next_state = UNLOCKING_1;}
            else if (sw_value == 0b1010) {next_state = UNLOCKING_2;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_2:
            if (sw_value == 0b1010) {next_state = UNLOCKING_2;}
            else if (sw_value == 0b1000) {next_state = UNLOCKING_3;}
            else {next_state = UNLOCKING_ERR;}
            break;
        case UNLOCKING_3:
            if (sw_value == 0b1000) {next_state = UNLOCKING_3;}
            else if (sw_value == 0b0000)
            {// unlocking sequence complete. unlocking the buttons
#ifdef DEBUG
                printf("unlocking sequence complete! unlocking the buttons\n");
#endif
                // Enable GPIO interrupts
                XGpio_InterruptEnable(&BTNInst, BTN_INT);
                next_state = UNLOCKED;
            }
            else {next_state = UNLOCKING_ERR;}
            break;
        case LOCKING_ERR:
            if (sw_value == 0b0000) {next_state = UNLOCKED;}
            else {next_state = LOCKING_ERR;}
            break;
        case UNLOCKING_ERR:
            if (sw_value == 0b1111) {next_state = LOCKED;}
            else {next_state = UNLOCKING_ERR;}
        }
#ifdef DEBUG
        if (next_state != current_state) {printf("going to %s\n", StateToName(next_state));}
#endif
    }

    return 0;
}

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

int BoardInit()
{
    int status;
    //----------------------------------------------------
    // INITIALIZE THE PERIPHERALS & SET DIRECTIONS OF GPIO
    //----------------------------------------------------
    // Initialise LEDs
    status = XGpio_Initialize(&LED_SW_Inst, LEDS_DEVICE_ID);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Initialise Push Buttons
    status = XGpio_Initialize(&BTNInst, BTNS_DEVICE_ID);
    if(status != XST_SUCCESS) return XST_FAILURE;
    // Initialize Switches
    // Set LEDs direction to outputs
    XGpio_SetDataDirection(&LED_SW_Inst, LEDS_CHANNEL, 0x00);
    // Set all buttons direction to inputs
    XGpio_SetDataDirection(&BTNInst, BTNS_CHANNEL, 0xFF);
    // Set all switches to inputs
    XGpio_SetDataDirection(&LED_SW_Inst, SW_CHANNEL, 0xFF);
    //----------------------------------------------------
    // SETUP THE TIMER
    //----------------------------------------------------
    status = XTmrCtr_Initialize(&TMRInst, TMR_DEVICE_ID);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XTmrCtr_SetHandler(&TMRInst, TMR_Intr_Handler, &TMRInst);
    XTmrCtr_SetResetValue(&TMRInst, 0, TMR_LOAD);
    XTmrCtr_SetOptions(&TMRInst, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

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

    XTmrCtr_Start(&TMRInst, 0);
    return status;
}

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, XTmrCtr *TmrInstancePtr, XGpio *GpioInstancePtr)
{
    XScuGic_Config *IntcConfig;
    int status;

    // Interrupt controller initialisation
    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;


    // Connect timer interrupt to handler
    status = XScuGic_Connect(&INTCInst,
                             INTC_TMR_INTERRUPT_ID,
                             (Xil_ExceptionHandler)TMR_Intr_Handler,
                             (void *)TmrInstancePtr);
    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);
    XScuGic_Enable(&INTCInst, INTC_TMR_INTERRUPT_ID);
    return XST_SUCCESS;
}
```