

# Final Project Report for Simple Minute Timer with Count Up and Count Down Feature

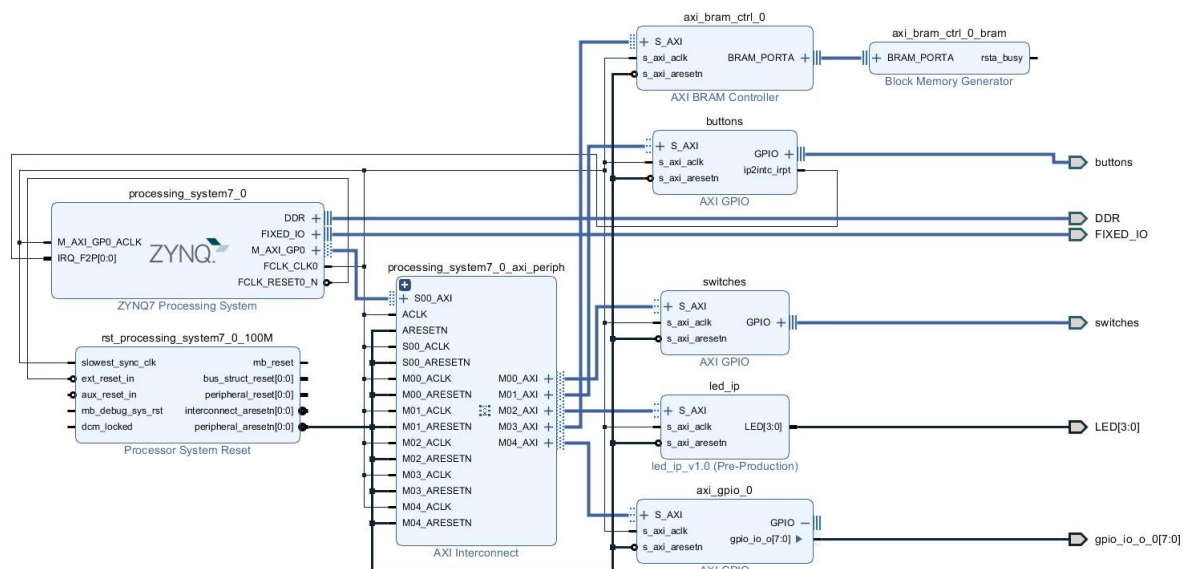
Matias Donato Soto Ricca - 6113218

**YouTube Link to Project Demonstration:** <https://youtu.be/E5Hessfqf0o>

## Final Project Prototype Design and Specifications

The final prototype is called the Minute Timer. Its main purpose is to serve as a dynamic timer, capable of serving as both a timer and a stopwatch. The user will also be able to manually stop the timer whenever desired and reset it to use it as many times as needed.

The block design consists of 4 AXI GPIOs, one for each of the 4 peripherals, which include the Seven Segment Display, Buttons, Switches, and LEDs. An interrupt was created for the buttons GPIO to allow an interrupt service routine to function accordingly with button presses. The buttons peripheral was chosen to contain an interrupt service routine because the peripheral is static, this means they do not get called consistently, such as the timer reset and the program exit button.



*Fig 1. Block Design of the Minute Timer*

As for the actual functionalities of the finalized project, they can be seen in Fig 2, with a new function called Timer Reset, which has been programmed and included. This new function allows the user to reset the timer back to 0. That will allow the user to be able to re-use the timer without the need to terminate the program and start it up all over again. The remaining functionalities have stayed the same since our initial proposal and brainstorming, with the leftmost switch counting up the timer by 1 for every elapsed second, the second-to-leftmost switch counting down, and the leftmost button terminating the program.

Four peripherals are used in the finalized version of the Minute Timer. These include: the LEDs, which light up when either of the two switches are active, the switches themselves, the

buttons, and the seven segment display. The buttons and switches are what allow the timer to be manipulated to the user's liking, and the seven segment display simply displays the value.

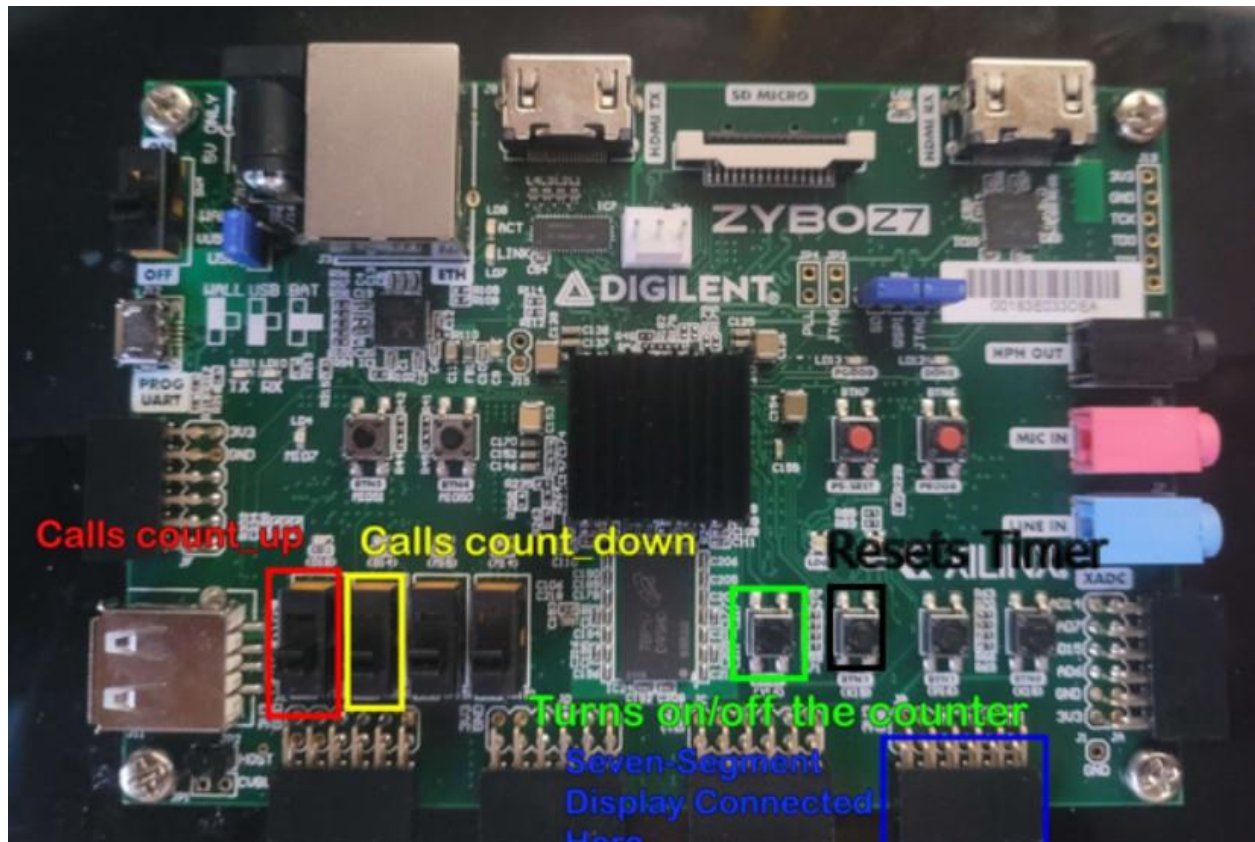


Fig 2. Functionalities of Minute Timer

Moving away from the hardware side of the project, we will go over and explain the various functionalities found within the source code of the project, as well as its functions, purpose, and logic behind it.

```
#include "xparameters.h"
#include <stdio.h>
// #include "platform.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xscutimer.h"
#include "led_ip.h"

//=====

XGpio output, dip, push;
int i, psb_check, dip_check, leftDigit, rightDigit;
XScuTimer Timer;
void displayVal(int, XGpio);
void delay(int);
int getAddr(int, int, XGpio);
static int Init_Peripherals();
static void Push_Intr_Handler(void *CallBackRef);
void getPeripheralStatus();
void init_Button_Int();
XScuGic IntcInstance; /* Interrupt Controller Instance */
XScuGic *IntcInstancePtr = &IntcInstance;
XScuGic_Config *IntcConfig;
```

Fig 3. Included header files, with function and variable declarations

As seen in Fig 3, we have the header files and declarations. For the sake of brevity, we will be discussing the most important of these. Xscugic.h allows for interrupts to be controlled by

the CPU, xgpio.h allows for GPIOs to be programmed and function in accordance with the program, and xparameters.h allows for us to quickly find and set the appropriate addresses for the peripherals.

```
static int Init_Peripherals()
{
    int pResult = XGpio_Initialize(&output, XPAR_SWITCHES_BASEADDR);
    if(pResult != XST_SUCCESS) {
        xil_printf("Failed to initialize Seven Segment Display\r\n");
        return XST_FAILURE;
    }

    pResult = XGpio_Initialize(&dip, XPAR_SWITCHES_DEVICE_ID);
    if(pResult != XST_SUCCESS) {
        xil_printf("Failed to initialize Switches\r\n");
        return XST_FAILURE;
    }
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    pResult = XGpio_Initialize(&push, XPAR_BUTTONS_DEVICE_ID);
    if(pResult != XST_SUCCESS) {
        xil_printf("Failed to initialize Buttons\r\n");
        return XST_FAILURE;
    }
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    return XST_SUCCESS;
}

void init_Button_Int() { //enables button interrupt service routine
    IntcConfig = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
    XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig, IntcConfig->CpuBaseAddress);
    XScuGic_SetPriorityTriggerType(IntcInstancePtr, XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR, 0xA0, 0x3);
    XScuGic_Connect(IntcInstancePtr, XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR, (Xil_ExceptionHandler)Push_Intr_Handler, (void *)&push);
    XScuGic_Enable(IntcInstancePtr, XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR);
    XScuGic_CPUWriteReg(IntcInstancePtr, 0x0, 0xf);
    XGpio_InterruptEnable(&push, 0xF);
    XGpio_InterruptGlobalEnable(&push);
}

static void Push_Intr_Handler(void *CallBackRef)
{
    XGpio *push_ptr = (XGpio *) CallBackRef;

    //xil_printf("Insider Button ISR! ... \r\n");

    // Disable GPIO interrupts
    XGpio_InterruptDisable(push_ptr, 0xF);

    //xil_printf("Here again ... \r\n");

    LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, 0xAAAA);

    (void)XGpio_InterruptClear(&push, 0xF);
    // Enable GPIO interrupts
    XGpio_InterruptEnable(&push, 0xF);
}
```

*Fig 4. Initializations and Interrupt Handler*

Fig 4 depicts the code responsible for initializing the peripherals and the interrupt service routine for the buttons, as well as the interrupt handler for a button press. The Init\_Peripherals() function is responsible for initializing one by one the button, switch, and seven segment display peripherals, making sure they have been initialized successfully. The init\_Button\_Int() function is responsible for going through the process of initialize the ISR for the button, which will allow an interrupt to properly occur and be handled by the CPU without any complications. The last function depicted is simply handling the interrupt once it occurs.

```

void displayVal(int i, XGpio o) {
    int leftDigit = i/10;
    int rightDigit = i % 10;

    if(i > 9) {
        for(int h = 0; h < 2000000; h++) {
            getAddr(leftDigit, 1, o);
            getAddr(rightDigit, 0, o);
        }
    }
    else {
        getAddr(rightDigit, 0, o);
        delay(50000000);
    }
}

int getAddr(int i, int r, XGpio o) {
    if(i < 0 || i > 9 ) { xil_printf("Error\n\r"); return 0; }
    u32 baseAddr = 0b00000000;
    if(r == 1) baseAddr |= 0b10000000;
    if(i == 0) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b00111111);
    }
    else if(i == 1) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b00000110);
    }
    else if(i == 2) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01011011);
    }
    else if(i == 3) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01001111);
    }
    else if(i == 4) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01100110);
    }
    else if(i == 5) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01101101);
    }
    else if(i == 6) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01111101);
    }
    else if(i == 7) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b00000111);
    }
    else if(i == 8) {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01111111);
    }
    else {
        XGpio_DiscreteWrite(&o, 1, baseAddr | 0b01100111);
    }
    return 1;
}

void delay(int t) {
    for(int i = 0; i < t*2; i++) {}
}

```

*Fig 5. Display Functions*

*Fig 5* depicts the functions responsible for actually displaying the current value onto the seven segment display. Due to the nature of the display being able to only display one digit at a time, the only way to display both at the same time is to loop through both digits constantly, which is less efficient than what we would have liked, but it completes its task successfully. Important note, the third bit in the u32 address is responsible for which digit to light up, with 1 being the left, and 0 being the right. We handled this by BITWISE OR-EQUALS the address with the inputted r value to determine which digit to light up, saving us from using more if else if statements.

```
void getPeripheralStatus() {  
    psb_check = XGpio_DiscreteRead(&push, 1); //button  
    dip_check = XGpio_DiscreteRead(&dip, 1); //switch  
    LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);  
}
```

*Fig 6. getPeripheralStatus function*

*Fig 6* simply shows the function responsible for constantly checking the status of both the switches and the buttons, as well as writing to the LED in case either switch is active, or both.

```

int main (void)
{
    Init_Peripherals();
    xil_printf("Peripherals Initialized Successfully...\r\n");

    init_Button_Int();
    xil_printf("Button ISR Initialized...\r\n");

    xil_printf("-- Start of the Program --\r\n");
    displayVal(i, output);

    int j = 0; //Time elapsed and value to be outputted to S.S.D

    while (1)
    {
        getPeripheralStatus();

        if(psb_check == 8) { xil_printf("Exiting Program\n\r"); break; }

        if(psb_check == 4) {
            xil_printf("Resetting timer...\n\r");
            j = 0;
            displayVal(j, output);
        }

        if(dip_check == 8 && j < 60) {
            xil_printf("Starting Count Up Sequence...\n\r");
            //countUp function
            while(j < 60 && dip_check == 8) {
                ++j;
                getPeripheralStatus();
                displayVal(j, output);
            }
        }
        if(dip_check == 4 && j > 0) {
            xil_printf("Starting Count Down Sequence...\n\r");
            //countDown function
            while(j > 0 && dip_check == 4) {
                --j;
                getPeripheralStatus();
                displayVal(j, output);
            }
        }

        while((dip_check == 12 || dip_check == 0) && j > 9 && psb_check == 0) {
            getPeripheralStatus();
            getAddr(j/10, 1, output);
            getAddr(j%10, 0, output);
        }

        for (i=0; i<99999999; i++);
    }
    //disables interrupt
    XGpio_InterruptDisable(&push, 0x3);
    XScuGic_Disable(IntcInstancePtr, XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR);
    XScuGic_Disconnect(IntcInstancePtr, XPAR_FABRIC_BUTTONS_IP2INTC_IRPT_INTR);
}

```

*Fig 7. Main body of the program*



In regards to *Fig 7*, the main body is responsible for various things: Initializing the ISR and peripherals, constantly checking for an input by either the switches or buttons, displaying the current value to the timer, and disabling the interrupt once the program is terminated.

### Analysis of Results and Tests

To ensure that our final prototype met the standards we have set for ourselves, as well as the objectives and overall functionality, we ran it through various tests to ensure proper quality. The grand majority of these tests were performed live on camera, which is displayed on the youtube link found in the submission to this assignment. These tests are listed below in, at least to our reasoning, increasing complexity:

- *Program starts up, with peripherals and interrupt service routine successfully initialized*

```
Peripherals Initialized Successfully...  
Button ISR Initialized...  
-- Start of the Program --
```

**Status:** Success

- *On start up, 0 is displayed on the seven segment display*



**Status:** Success

- *The timer does not change when neither switch is active*

**Status:** Success (Proven in the video)

- *The timer does not change when Count Down switch is active and timer is at 0*

**Status:** Success (Proven in the video)

- *The timer does not change when Count Up switch is active and timer is at 60*

**Status:** Success (Proven in the video)

- *The timer does not change when both switches are active*

**Status:** Success (Proven in the video)

- *The timer properly counts up to 60 as long as Count Up switch is active*

**Status:** Success (Proven in the Video)

- *The timer properly counts down from 60 to 0 as long as Count Down switch is active*

**Status:** Success (Proven in the video)

- *The appropriate value is displayed when timer is stopped mid-way through either Count Up or Count Down*

**Status:** Success (Proven in two separate occasions in the video)


- *The reset button resets the timer back to 0, regardless of its current value*

```
-- Start of the Program --  
Starting Count Up Sequence...  
Resetting timer...
```

**Status:** Success (Also proven in the video)

- *The terminate button successfully terminates the program*

```
Resetting timer...  
Exiting Program
```



**Status:** Success

## Conclusions

To conclude the Minute Timer, the overall progression while designing it went smoother than expected. We did run into a few obstacles, most notably getting the time elapsed to display on the seven segment display and properly setting up the interrupt sequence for button presses. The block design went very well with no major issues, and programming the timer.c source code went fluently. We learned a lot about the functionalities behind each of the peripherals and how to manipulate them via code to function in accordance with what we want. The timer also functions exactly as we hoped it to work, with no major bugs or memory leaks.

## Future Work

While the current design of the Minute Timer is very simple, it can easily be used as a base for more complex works regarding the use of the seven segment display. It could potentially be connected to other external peripherals, with expanded functionalities and more intuitive uses. There is boundless potential for our project, with the possibilities being endless. We hope to also continue developing this project, expanding its capabilities and adding more functions, allowing its uses to broaden, and for the Minute Timer to reach the hands of even more people.

## Additional Documents

The readme.md can be found within the project files.