FreeRTOS Data Queue on Zybo v2

Robert Bara
Robert.Bara@temple.edu

Summary

Lab 5 introduces the concept of data queues in FreeRTOS by sending data from the input buttons and switches of the Zybo board to be encoded, sent to a queue, and displayed across the output LEDs.

Introduction

To introduce the concept of data queues, two tasks will be created within FreeRTOS:

TaskBTNSW reads inputs from push buttons and switches at the idle task priority+1. The input button contains a proper button debounce, and upon completion if the switch value matches the button value, then both values are combined into an encoded value to be sent to the queue, with a block time of 200 seconds. This is done by creating an 8bit binary number where the switch values hold the upper half of the binary value and the lower half holds the push button values. That is to say, BTN0=1, BTN1=2, BTN2=4, BTN3=8, and SW0=16, SW1=32, SW2=64, SW3=128. An example of the encoded value will sum the button and switch inputs, so if both SW0 and Btn0 are high, BTN=4b0001, SW=4b0001, and switch gets multiplied by 16 and added to Btn to create an encoded value=8b0001 0001.

TaskLED holds an idle task priority+2 and receives an encoded 8bit binary value. In the case of the example mentioned above, LED0 will light up since the LEDs can be examined as a 4bit binary number 4b0001. This receives the encoded value from the *xQueueBtnSw* every 5 seconds and holds a block time of 60 seconds to receive. Upon receiving data, the corresponding binary value should display across the appropriate LED for 2 seconds. If the inputs from *TaskBTNSW* were invalid, nothing is sent to the queue and an error message is shown flashing the LEDs across for 1 second on/off.

Discussion

Hardware Design

Lab 5's hardware design is a straightforward design consisting of the Processor and 2 GPIOs. Begin by creating a block diagram in Vivado that adds the Zynq processor in and run the automation. Next add GPIO0 which will map two inputs: channel 1-btns, channel 2-sw. Then add a second GPIO1 to map LEDs as outputs. Run the automation and ensure the reset and axi_periph blocks are generated and wired correctly to the processor and GPIOs. Verify the design, create the HDL wrapper, and generate a bitstream. Upon completion, export the design to hardware and launch SDK to begin the FreeRTOS software design. The figure below examines the completed block diagram:

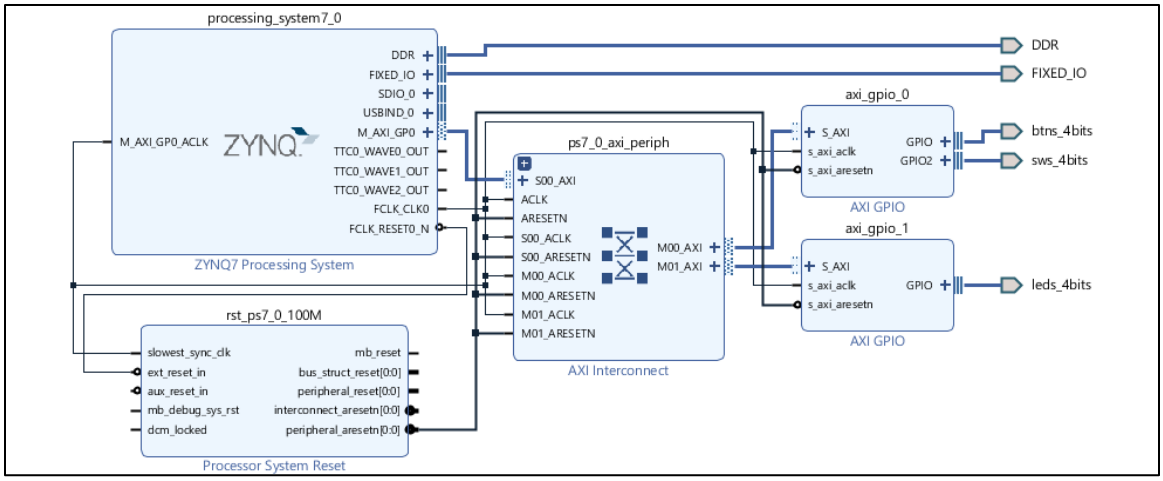

Figure 1. Completed Hardware Design

FreeRTOS Program

The FreeRTOS program can be implemented by modifying the existing *freertos_hello_world.c* file, while removing any unnecessary parameters and functions such as the timer.

Initialization

The program initialization begins by defining the appropriate FreeRTOS and hardware headings, defining *taskLED* and *taskBTNSW* with their respective handles and Queue that will send and receive data.

```
    Copyright (C) 2017 Amazon.com, Inc. or its affiliates.  All Rights Reserved.
//Robert Bara Lab 5 ECE3623 Spring 2021
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
/* Xilinx includes. */
#include "xil_printf.h"
#include "xparameters.h"
#include "xgpio.h"

//definitions
#define printf xil_printf
/*-----------------------------------------------------------*/
/* The tasks as described at the top of this file. */
static void taskLED( void *pvParameters );
static void taskBTNSW( void *pvParameters );
/*-----------------------------------------------------------*/
/* Defining Task Handles */
static TaskHandle_t xTaskLED;
static TaskHandle_t xTaskBTNSW;
/* The queue used by the taskLED, taskBTN, taskSW tasks, as described at the top of this
file. */
static QueueHandle_t xQueueBtnSw = NULL;
```

Figure 2. Defining Headers and Parameters

The program then defines the hardware GPIOs and their channels. If the GPIOs are initialized correctly, then LEDs are set as outputs, while buttons and switches are mapped as inputs:

```
//GPIO definitions
#define LED_DEVICE_ID XPAR_AXI_GPIO_1_DEVICE_ID //GPIO device connected to LED
#define INP_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID //GPIO device connected to BTNs and SW
XGpio LEDInst, InpInst;
//Defining each channel
#define LED_CHANNEL 1
#define BTN_CHANNEL 1
#define SW_CHANNEL 2

//Main Function
int main( void ){
    //GPIO Driver Initialization
    int status;
    status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);             // LED initialization
    if (status != XST_SUCCESS) {printf("LED Initialization failed"); return XST_FAILURE;}
    XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x00);            //Setting LED to Output

    //Input Initialization
    status = XGpio_Initialize(&InpInst, INP_DEVICE_ID);
    if (status != XST_SUCCESS) {printf("Input Initialization failed"); return XST_FAILURE;}
    //Setting Btn and SW as Inputs
    XGpio_SetDataDirection(&InpInst, BTN_CHANNEL, 0xFF);
    XGpio_SetDataDirection(&InpInst, SW_CHANNEL, 0xFF);
    XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,0); //initializing LED as off
    printf("Initialization Complete\n");
```

Figure 3. Mapping Hardware to Software

If initialization is completed, the respective *taskLED* and *taskBTNSW* are created with proper comments shown below. *QueueBtnSw* is created as well to hold 5 items within the queue. The items being the encoded sent by *taskBTNSW*, which will be received, decoded, and outputted to the LEDs within *taskLED*. The initialization of the queue is then checked and if the queue is created, then the tasks start running:

```
/* Creating the two tasks.  The taskLED task is given a lower priority than the
taskBTNSW task, so the taskBTNSW task will leave the Blocked state and pre-empt the taskLED
task as soon as the taskBTNSW task places an item in the queue. */
xTaskCreate(    taskLED,                    /* The function that implements the task. */
                ( const char * ) "L",       /* Text name for the task, provided to assist debugging only. */
                configMINIMAL_STACK_SIZE,   /* The stack allocated to the task. */
                NULL,                       /* The task parameter is not used, so set to NULL. */
                tskIDLE_PRIORITY + 2,       /* The task runs at the idle priority. */
                &xTaskLED );                /* Address of handler. */

xTaskCreate( taskBTNSW,                     /* The function that implements the task. */
             ( const char * ) "BSW",        /* Text name for the task, provided to assist debugging only. */
             configMINIMAL_STACK_SIZE,      /* The stack allocated to the task. */
             NULL,                          /* The task parameter is not used, so set to NULL. */
             tskIDLE_PRIORITY + 1,          /* The task runs at the idle priority. */
             &xTaskBTNSW );                 /* Address of handler. */

    /* Create the queue used by the tasks. */
    xQueueBtnSw = xQueueCreate(     5,                      /* There are 5 spaces in the queue. */
                               sizeof( int ) );    /* Each space in the queue is large enough to hold a uint32_t. */

    /* Check the queue was created. */
    configASSERT( xQueueBtnSw );

/* Start the tasks and timer running. */
vTaskStartScheduler();

/* If all is well, the scheduler will now be running, and the following line
will never be reached.  If the following line does execute, then there was
insufficient FreeRTOS heap memory available for the idle and/or timer tasks
to be created.  See the memory management section on the FreeRTOS web site
for more details. */
for(;;){
}
}
```

Figure 4. Creating Tasks and Queue

Subroutines (FreeRTOS Tasks)

Examining the tasks of this program, I will start by discussing the input task and then discuss how the output task functions.

taskBTNSW

Since this task has a block time of 200 seconds, a delay is defined. Additionally, a delay of 0.4 seconds is defined to perform button debouncing. Using the local the variables, the *taskBTNSW* begins by reading switches and buttons. Button debouncing is accomplished by storing the button input into a temporary value, and after a 0.4 second delay, button is read again, and another delay is generated. To ensure a valid input, an if statement runs only when button and the debounce value are equal and not read as 0 (the value when nothing is pressed). If this is true, the encoding process begins. The input data will be encoded as one 8bit binary number, so the switch input gets multiplied by 16 so it will represent the upper 4bits of the 8bit number. The encoded data variable is then the sum of the switch input and button input. The data is then sent to the queue and checks for available spaces left in the queue. If the queue is full, it prints an error message, and yields a block time of 200 seconds, unless if the queue has an available space that can be filled.

```
static void taskBTNSW( void *pvParameters )
{
int btn, sw;
int debounce, data;
const TickType_t x200second = pdMS_TO_TICKS( 200000UL );//initialize 200 second Block Time to transmit data
const TickType_t x0_4second=pdMS_TO_TICKS(400uL ); //initialize 0.4 second delay for debouncing
    while(1)
    {
        //Reading Inputs from Board
        btn=XGpio_DiscreteRead(&InpInst,BTN_CHANNEL);

        //store value debouncing
        debounce=btn;
        /* Delay for 0.4second. */
        vTaskDelay(x0_4second);
        //read button again
        btn=XGpio_DiscreteRead(&InpInst,BTN_CHANNEL);

        //If Switch and Button combinations match, Encode and Send to the Queue
        if (btn==debounce && debounce!=0)
        {   sw=XGpio_DiscreteRead(&InpInst,SW_CHANNEL);
            sw=sw*16; //Encoding Sw to be higher bits
            data=btn+sw; //Combining inputs as an 8bit binary
                /* Send the next value on the queue.  The queue should always be
                   empty at this point so a block time of is used. */
                xQueueSend( xQueueBtnSw,            /* The queue being written to. */
                            &data,                   /* The address of the data being sent. */
                            x200second );           /* 200 sec block time. */
                printf("Sent: btn=%d\tsw=%d\tSpaces in Q=%d\n",btn, sw,uxQueueSpacesAvailable(xQueueBtnSw));
            if(uxQueueSpacesAvailable(xQueueBtnSw)==0)
            {
                printf("Queue is Full\n");
            }
        }
    }
}
```

Figure 5. TaskBTNSW

taskLED

TaskLED begins by defining the block times used, as well as the delay used when flashing values to LEDs. LED is initialized as zero, and local variables are created. The data being fed to the LED is checked and this will determine the block time for each conditional. Status is also defined as whether *taskLED* is receiving from the queue or not.

```
/*-----------------------------------------------------------*/
static void taskLED( void *pvParameters )
{

    const TickType_t x1second = pdMS_TO_TICKS( 1000UL ); //initialize 1 second task delay
    const TickType_t x2second = pdMS_TO_TICKS( 2000UL ); //initialize 2 second task delay for LEDs
    const TickType_t x5second = pdMS_TO_TICKS( 5000UL );//initialize 5 second to receive items from queue
    const TickType_t x60second = pdMS_TO_TICKS(60000UL); //initialize 60 second block time when receiving
    int block_time=x1second;//initializing block_time to 1 sec
    int error=0b1111;
    int btn=0, sw=0, sw4bit=0,led=0,inp=0,status=0;
    while(1)
    {
        if(led!=error||led!=~error||led!=0) block_time=x1second;//When nothing is read from the Queue, Block Time is 1 sec
        if((led==btn||led==0) && btn!=0) block_time=x60second;//After reading from the Queue, Block Time is 60 sec

        //Status will execute if the queue is being read
        status=xQueueReceive(xQueueBtnSw, &led, block_time);
```

Figure 6. Initializing TaskLED, Deciding BlockTimes

If there is no data being received by the queue, an error sequence is accomplished by flashing LEDs and printing in the terminal that the queue is empty.

```
/*if nothing is being read from the queue */
if(status==0)
{   btn=0;
    inp=0;
    led=error;
    //Flash LEDs for 1 second when Queue is empty
    printf("Queue is Empty\n");
    XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
    error=~error;
    vTaskDelay(x1second);
}
```

Figure 7. When nothing is being received, flash LEDs

When the queue receives data, the data is stored to the address of led for decoding and displaying. The switch and button inputs can be decoded by undoing the math that was used to convert them into an 8bit binary number. This is done by first finding the 4bit input of the switches by masking the lower 4bits of the encoded binary data using a right shift operator. Button is found similarly by masking the upper 4bits of the 8bit binary value by comparing the encoded value with an AND bitwise operand and the hex value 0x0F or 4b1111. Switch based on the encoding process, is then decoded as the encoded value minus the button value. LEDs are only displayed if the 4bit switch and button inputs were matching. If the inputs are a match, the output displayed across the LEDs is the lower 4bits of the 8bit encoded binary number, or the sum of both inputs. This is displayed for 2seconds, then turned off for 2 seconds before writing the next item to the queue. If the queue is not filled correctly, then it will enter a block time of 60 seconds before emptying the queue.

```
        /* If Data is in Queue, decode and print to LEDs, Block Time of 60sec */
        else if(status==1)
        {

            XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,0);//initialize LEDs
            //Read from the Queue every 5 seconds
            vTaskDelay(x5second);

            //Decoding Values from Queue
            sw4bit=led>>4;//Mask lower bits of encoded value to find switch input
            btn=led&0x0F; //Using AND operand to compare encoded value with 0b1111, therefore masking upper bits
            sw=led-btn; //Finding out what switch is based on using the upper 4bits of the 8bit binary
            inp=sw4bit&btn;//Comparing btn and sw inputs
            printf("Decoded: btn=%d, sw=%d, sw input was=%d\tSpaces left in Q=%d\n",btn, sw, sw4bit,uxQueueSpacesAvailable(xQueueBtnSw));

            //Data may be sent and read to the queue, but only when the sw and btn inputs match will the LED output the btn and sw initial input
            if(inp==btn){
                //If a valid input is delivered, Write 4bit input to LED for 2 seconds
                led=inp;
                printf("Valid Input led=%d\n",led);
                XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
                vTaskDelay(x2second);

                //Waiting turning off for 2 seconds to load in Next Queue
                led=0;
                XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
                vTaskDelay(x2second);

            }
        }
    }
}
```

Figure 8. If Data was sent, receive it and decode it

Verification

Video Link:

https://youtu.be/jhJ8pZOzKl4 Please forgive the fact I accidentally say lab 6 when introducing the video, I did not realize I misspoke until after I uploaded the video.

Further verification can be seen as follows by comparing the LEDs to SDK terminal:

When Btn and Sw inputs do not match, the data is decoded but LEDs are not lit up, in this case btn 2 was pressed.

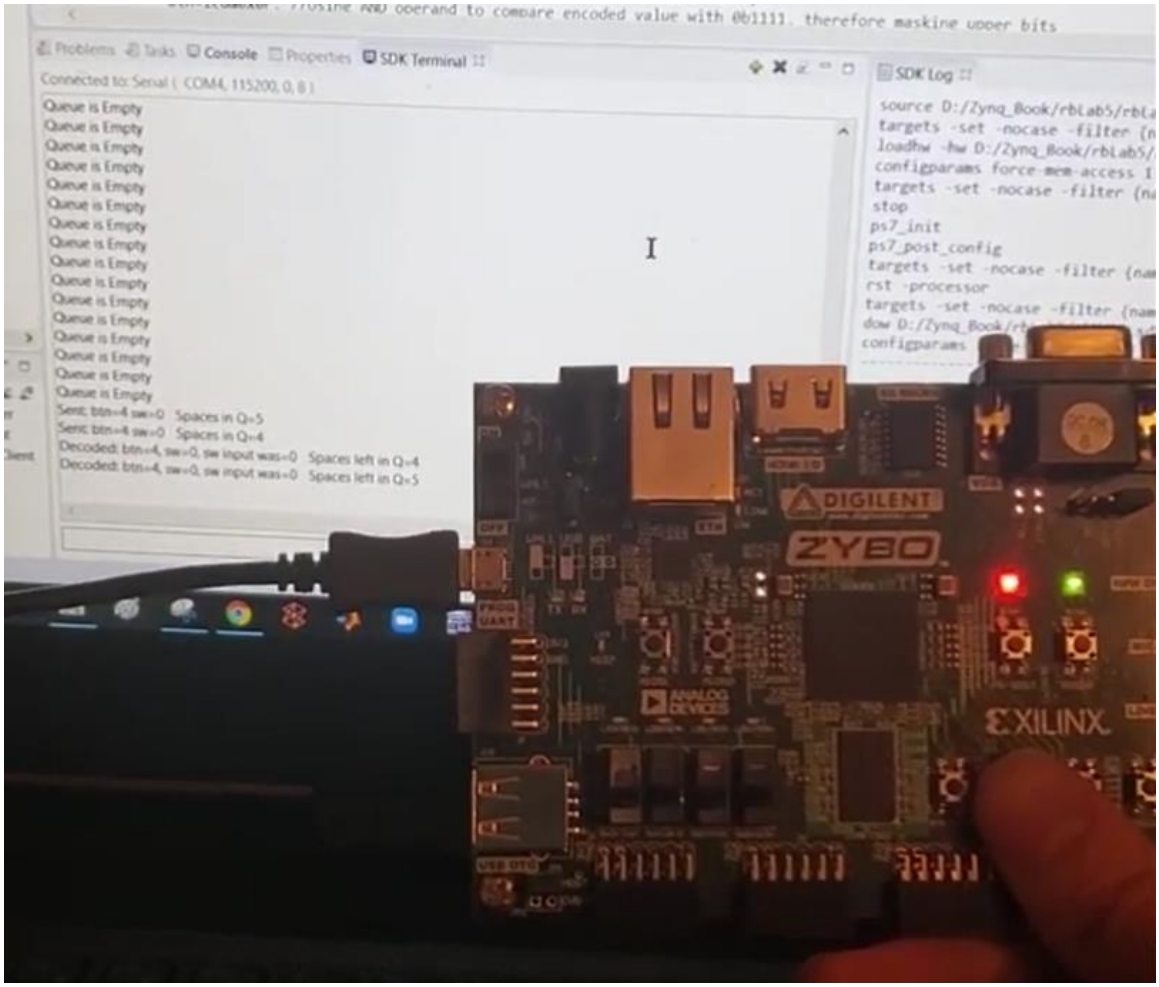

Figure 9. Inputs do not match

An example of filling the Queue, decoding data, then exiting the block state by filling the queue up when 2 slots are available and using multiple sw/btns as inputs. This particular example uses btn0 and btn1 are inputs while sw0 and sw1 are high, there fore LEDs 0 and 1 are high:

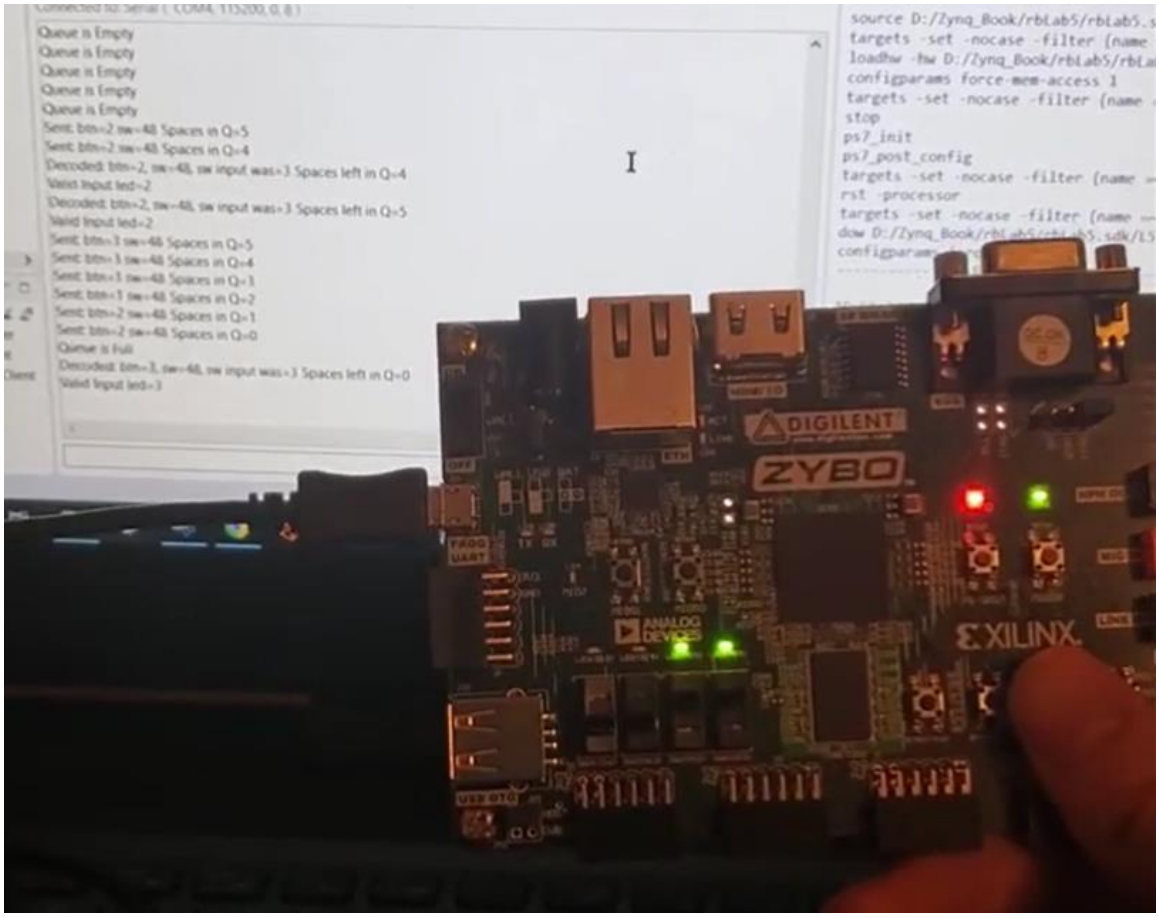

Figure 10. Multiple Buttons, full queue (block time), and displaying multiple btns

Letting the queue empty and returning to the LED flash error state with a block time of 1 second:

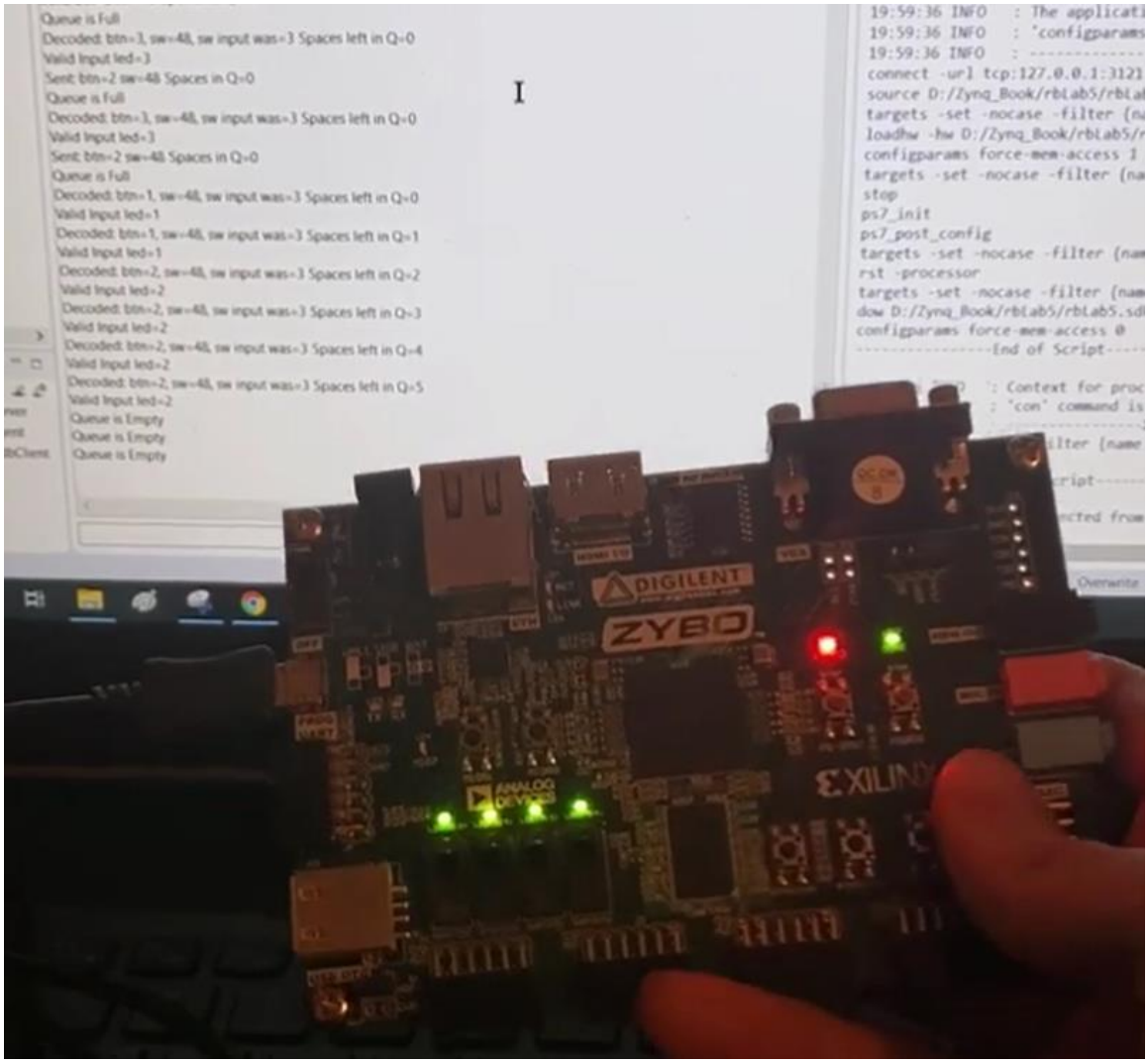

Figure 11. Emptying Queue, flashing LEDs

Conclusion

Overall, I thought this lab was one of the easier labs we have completed within this course. The concept of queue management was a little bit tricky since at first, I was only able to receive every other queue element, but Zach pointed out my issue, realizing my conditionals within *taskLED* were reading from the queue twice even when there was no data. After a bit of debugging, I was able to successfully encode and decode the inputs using shifts and bitwise masking, and light up each LED depending on a valid input and the size of the queue.

Appendix

FreeRTOS Code

```
/*
    Copyright (C) 2017 Amazon.com, Inc. or its affiliates.  All Rights
Reserved.
    Copyright (C) 2012 - 2018 Xilinx, Inc. All Rights Reserved.

    Permission is hereby granted, free of charge, to any person obtaining a
copy of
    this software and associated documentation files (the "Software"), to deal
in
    the Software without restriction, including without limitation the rights
to
    use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of
    the Software, and to permit persons to whom the Software is furnished to
do so,
    subject to the following conditions:

    The above copyright notice and this permission notice shall be included in
all
    copies or substantial portions of the Software. If you wish to use our
Amazon
    FreeRTOS name, please do so in a fair use way that does not cause
confusion.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS
    FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR
    COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER
    IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

    http://www.FreeRTOS.org
    http://aws.amazon.com/freertos


    1 tab == 4 spaces!
*/
//Robert Bara Lab 5 ECE3623 Spring 2021
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
/* Xilinx includes. */
#include "xil_printf.h"
#include "xparameters.h"
#include "xgpio.h"

//definitions
#define printf xil_printf
```

```
/*-----------------------------------------------------------*/
/* The tasks as described at the top of this file. */
static void taskLED( void *pvParameters );
static void taskBTNSW( void *pvParameters );
/*-----------------------------------------------------------*/
/* Defining Task Handles */
static TaskHandle_t xTaskLED;
static TaskHandle_t xTaskBTNSW;
/* The queue used by the taskLED, taskBTN, taskSW tasks, as described at the
top of this
file. */
static QueueHandle_t xQueueBtnSw = NULL;

//GPIO definitions
#define LED_DEVICE_ID XPAR_AXI_GPIO_1_DEVICE_ID //GPIO device connected to LED
#define INP_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID //GPIO device connected to
BTNs and SW
XGpio LEDInst, InpInst;
//Defining each channel
#define LED_CHANNEL 1
#define BTN_CHANNEL 1
#define SW_CHANNEL 2

//Main Function
int main( void ){
	//GPIO Driver Initialization
	int status;
	status = XGpio_Initialize(&LEDInst, LED_DEVICE_ID);		// LED
initialization
	if (status != XST_SUCCESS) {printf("LED Initialization failed"); return
XST_FAILURE;}
	XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x00);
	//Setting LED to Output

	//Input Initialization
	status = XGpio_Initialize(&InpInst, INP_DEVICE_ID);
	if (status != XST_SUCCESS) {printf("Input Initialization failed");
return XST_FAILURE;}
	//Setting Btn and SW as Inputs
	XGpio_SetDataDirection(&InpInst, BTN_CHANNEL, 0xFF);
	XGpio_SetDataDirection(&InpInst, SW_CHANNEL, 0xFF);
	XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,0); //initializing LED as off
	printf("Initialization Complete\n");

	/* Creating the two tasks.  The taskLED task is given a lower priority
than the
	taskBTNSW task, so the taskBTNSW task will leave the Blocked state and
pre-empt the taskLED
	task as soon as the taskBTNSW task places an item in the queue. */
	xTaskCreate( taskLED,					/* The function
that implements the task. */
				( const char * ) "L",		/* Text name
for the task, provided to assist debugging only. */
				configMINIMAL_STACK_SIZE, /* The stack
allocated to the task. */
```

```
                                NULL,                                   /* The
task parameter is not used, so set to NULL. */
                                tskIDLE_PRIORITY + 2,           /* The task
runs at the idle priority. */
                                &xTaskLED );                    /* Address of
handler. */

      xTaskCreate( taskBTNSW,                                   /* The
function that implements the task. */
                           ( const char * ) "BSW",         /* Text name for
the task, provided to assist debugging only. */
                           configMINIMAL_STACK_SIZE,       /* The stack
allocated to the task. */
                           NULL,                                        /* The
task parameter is not used, so set to NULL. */
                           tskIDLE_PRIORITY + 1,                /* The task
runs at the idle priority. */
                           &xTaskBTNSW );                               /*
Address of handler. */

            /* Create the queue used by the tasks. */
            xQueueBtnSw = xQueueCreate(      5,
      /* There are 5 spaces in the queue. */
                                                      sizeof( int ) );    /*
Each space in the queue is large enough to hold a uint32_t. */

            /* Check the queue was created. */
            configASSERT( xQueueBtnSw );

      /* Start the tasks and timer running. */
      vTaskStartScheduler();

      /* If all is well, the scheduler will now be running, and the following
line
      will never be reached.  If the following line does execute, then there
was
      insufficient FreeRTOS heap memory available for the idle and/or timer
tasks
      to be created.  See the memory management section on the FreeRTOS web
site
      for more details. */
      for(;;){
}
}

/*-----------------------------------------------------------*/
static void taskLED( void *pvParameters )
{

      const TickType_t x1second = pdMS_TO_TICKS( 1000UL ); //initialize 1
second task delay
      const TickType_t x2second = pdMS_TO_TICKS( 2000UL ); //initialize 2
second task delay for LEDs
      const TickType_t x5second = pdMS_TO_TICKS( 5000UL );//initialize 5
second to receive items from queue
```

```
	const TickType_t x60second = pdMS_TO_TICKS(60000UL); //initialize 60
second block time when receiving
	int block_time=x1second;//initializing block_time to 1 sec
	int error=0b1111;
	int btn=0, sw=0, sw4bit=0,led=0,inp=0,status=0;
	while(1)
	{
		if(led!=error||led!=~error||led!=0) block_time=x1second;//When
nothing is read from the Queue, Block Time is 1 sec
		if((led==btn||led==0) && btn!=0) block_time=x60second;//After
reading from the Queue, Block Time is 60 sec

		//Status will execute if the queue is being read
		status=xQueueReceive(xQueueBtnSw, &led, block_time);
		/*if nothing is being read from the queue */
		if(status==0)
		{	btn=0;
			inp=0;
			led=error;
			//Flash LEDs for 1 second when Queue is empty
			printf("Queue is Empty\n");
			XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
			error=~error;
			vTaskDelay(x1second);
		}
		/* If Data is in Queue, decode and print to LEDs, Block Time of
60sec */
		else if(status==1)
		{

			XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,0);//initialize
LEDs
			//Read from the Queue every 5 seconds
			vTaskDelay(x5second);

			//Decoding Values from Queue
			sw4bit=led>>4;//Mask lower bits of encoded value to find
switch input
			btn=led&0x0F; //Using AND operand to compare encoded value
with 0b1111, therefore masking upper bits
			sw=led-btn; //Finding out what switch is based on using
the upper 4bits of the 8bit binary
			inp=sw4bit&btn;//Comparing btn and sw inputs
			printf("Decoded: btn=%d, sw=%d, sw input was=%d\tSpaces
left in Q=%d\n",btn, sw, sw4bit,uxQueueSpacesAvailable(xQueueBtnSw));

			//Data may be sent and read to the queue, but only when
the sw and btn inputs match will the LED output the btn and sw initial input
			if(inp==btn){
				//If a valid input is delivered, Write 4bit input to
LED for 2 seconds
				led=inp;
				printf("Valid Input led=%d\n",led);
				XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
				vTaskDelay(x2second);
```

```
                        //Waiting turning off for 2 seconds to load in Next
Queue
                        led=0;
                        XGpio_DiscreteWrite(&LEDInst,LED_CHANNEL,led);
                        vTaskDelay(x2second);

                }
        }
    }
}

/*-----------------------------------------------------------*/
static void taskBTNSW( void *pvParameters )
{
int btn, sw;
int debounce, data;
const TickType_t x200second = pdMS_TO_TICKS( 200000UL );//initialize 200
second Block Time to transmit data
const TickType_t x0_4second=pdMS_TO_TICKS(400uL ); //initialize 0.4 second
delay for debouncing
    while(1)
    {
        //Reading Inputs from Board
        btn=XGpio_DiscreteRead(&InpInst,BTN_CHANNEL);

        //store value debouncing
        debounce=btn;
        /* Delay for 0.4second. */
        vTaskDelay(x0_4second);
        //read button again
        btn=XGpio_DiscreteRead(&InpInst,BTN_CHANNEL);

        //If Switch and Button combinations match, Encode and Send to the
Queue
        if (btn==debounce && debounce!=0)
        {   sw=XGpio_DiscreteRead(&InpInst,SW_CHANNEL);
            sw=sw*16; //Encoding Sw to be higher bits
            data=btn+sw; //Combining inputs as an 8bit binary
                /* Send the next value on the queue.  The queue
should always be
                   empty at this point so a block time of is used.
*/
                xQueueSend( xQueueBtnSw,               /* The queue
being written to. */
                                        &data,
/* The address of the data being sent. */
                                        x200second );             /* 200
sec block time. */
                printf("Sent: btn=%d\tsw=%d\tSpaces in Q=%d\n",btn,
sw,uxQueueSpacesAvailable(xQueueBtnSw));
            if(uxQueueSpacesAvailable(xQueueBtnSw)==0)
            {
                printf("Queue is Full\n");
            }
```

```
            }
        }
}
/*-------------------------------------------------------------*/
```