ECE 315 Lab 2

Lora Ma
Benjamin Kong

ECE 315 Lab Section H41

March 3, 2022

Contents

1 Abstract

The purpose of this lab was to

- gain experience using an ASCII-encoded serial communications connection between a PC host and a Zybo Z7 microcomputer board
- gain experience with the UART interface on the Zynq-7000 SoC
- gain experience interfacing using hardware and polling interrupts
- gain experience using queues to decouple the execution of software tasks

We will be using the Zybo Z7 development board by Digilent. The board is built around the Xilinx Zynq-7010 System-on-Chip silicon chip and contains two 667 MHz ARM Cortex A9 32-bit CPUs. For this lab, we will only be using one of the CPUs, CPU0, to run the FreeRTOS real-time kernel. CPU1 will be left disabled.

In this lab, we will complete two exercises. In exercise 1, we will design a morse code decoder system using polled UART interfacing. The user may enter Morse code messages (up to 500 characters long) using only a dot, dash, and a vertical bar into the console terminal on the host PC over the serial connection. Then, this will be decoded back into its original alphanumeric message and sent back to the console. Once the user has finished typing their Morse code message, the user must press the Enter key, press the # key, and then press the Enter key once again. Then, the morse encoded message is processed by the FreeRTOS task and the decoded message is returned by the Zybo Z7 board over the serial connection and displayed on the PC console.

In exercise 2, we will be modifying and enhancing an interrupt-driven driver for UART1 in the Zynq-7000 SoC. The purpose of this exercise is to gain experience using the interrupt-driven method to transmit bytes to the SDK console through the same UART. We want to abstract the driver for the queues and provide a more efficient interrupt-driven operation in the receive and transmit directions. We are given a driver file where we need to fill out 4 driver functions (`MyIsReceiveData()`, `MyReceiveByte()`, `MyIsTransmitFull()`, and `MySendByte()`). Additionally, we want to change the capitalization of characters received from the SDK console and implement a mechanism that detects `"\r%\r"` to clear the variables `CountRxIrq`, `CountTxIrq`, and `Countbytes`. We also need a mechanism to detect `"\r#\r"` that prints three status messages

```
Number of bytes processed: 40
```

```
Number of Rx interrupts: 32
Number of Tx interrupts: 25
```

2 Design

2.1 Exercise 1 Design

Please refer to Appendix 6.1 for the completed source code and line numbers.

For this exercise, we have three tasks – `TaskMorseMsgReceiver`, `TaskMorseMsgProcessor`, and `TaskDecodedMsgTransmitter`. First, we create these tasks, initialize UART in normal mode, and create our queues `xQueue_12` and `xQueue_23`. `xQueue_12` is the queue between `TaskMorseMsgReceiver` and `TaskMorseMsgProcessor` while `xQueue_23` is the queue between `TaskMorseMsgProcessor` and `TaskDecodedMsgTransmitter`. In the task `TaskMorseMsgReceiver()`, we first calculate the cycle time period `xPeriod` and initialized the the time at which the task was last unblocked `xLastWakeTime` to the count of ticks since `vTaskStartScheduler` was called. Then, we have a for loop that runs for the duration of the task where we repeat the following:

1. Delay task for a period of `xPeriod`
2. Check if there's room in the `xQueue_12` queue
3. If there is room in the queue, we receive the UART character and send it the back of the queue

For `TaskMorseMsgProcessor`, we initialize a char array `read_from_queue12_value`, a variable `no_of_characters_read` for the number of characters read, a variable `break_loop` for when the "#̊̊" sequence is detected to break out of the while loop, and an integer variable for the `error_flag`. Then we have a for loop that initailizes the variables to 0 or False. Then we have a while loop that runs while we are within the 500 character limit and if an item was successfully received from the queue. Within this loop we

- increment the number of character read `no_of_characters_read`
- try to receive a char from `xQueue_12` and put it into the char array `read_from_queue12_value`
- check for the "#̊̊" sequence. If it is detected, we run the characters in the char array `read_from_queue12_value` through the `morseToTextConverter()` and break out of the while loop.

Then, we send the translated message to `xQueue_23`. If we overflowed, we send the error message.

For `TaskDecodedMsgTransmitter`, we first calculate the cycle time period `xPeriod` and initialized the time at which the task was last unblocked `xLastWakeTime` to the count of ticks since `vTaskStartScheduler` was called. Then, we have a for loop where:

- while we are unable to successfully receive something from the queue, we wait
- if we receive something, check if the transmitter is full. If it is, delay task for a period of `xPeriod`, otherwise, sent the byte to the UART and break out of the loop.

2.2 Exercise 2 Design

Please refer to Appendix 6.2 for the completed source code and line numbers.

The goal of this exercise is to use the interrupt-driven method to transmist bytes to the SDK console through the UART. In `uart_driver.h`, we completed some functions:

- `Interrupt_Handler`: if the event is `XUARTPS_EVENT_RECV_DATA`, we increment the receive interrupt counter variable. We then check for bytes from the UART and send it to the receive queue if there are bytes. Otherwise, if the event is `XUARTPS_EVENT_SENT_DATA`, we increment the transmit interrupt counter variable. We then read data from the transmit queue and send it to the UART.
- `MyIsReceiveData`: we return `pdTRUE` if there are messages waiting in the receive queue. Otherwise, we return `pdFALSE`.
- `MyReceiveByte`: we use `taskENTER_CRITICAL` and `taskEXIT_CRITICAL` since it is a critical section to read a value from the receive queue.
- `MyIsTransmitFull`: if the transmit queue is full, we return `pdTRUE`; otherwise we return `pdFALSE`.
- `MySendByte`: again we use `taskENTER_CRITICAL` and `taskEXIT_CRITICAL` since it is a critical section. We either use polling or insert to the queue. We use polling if the transmit FIFO is empty; otherwise we attempt to send to the transmit queue.

In `part2_lab2_main.c`, we have two tasks `uart_receive_task` and `uart_transmit_task`. We create two queues to handle communication between the two tasks. These two queues have a

maximum size of 100 as defined in the `initialization.h` file. The queues store variables of type `u8` (basically the queues will store characters).

`uart_receive_task`: this task first prompts the user to enter in the text. It then starts an infinite loop. Within the loop, we get the data (a character) and store it in a variable. We then invert the capitalization of the character. We then send the character to the UART. We may perform a context switch if there is a carriage return character. Otherwise, we attempt to detect the `\r%\r` sequence.

`uart_transmit_task`: when this task is running, it outputs information such as the number of bytes processed, number of Rx interrupts, and number of Tx interrupts. When it finishes, it gives up the CPU again.

3 Testing

3.1 Exercise 1 Tests

Table 1: Tests for exercise 1.

Description	**Expected**	**Result**	**Success**
...|-|.|...-|.|-.| .--|.-|...| -...|---|.-.|-.| ---|-.| .-|.--.|.-.|..|.-..| .----|--...|--..--| .----|----.|----.|---..|.-.-.-|	STEVEN WAS BORN ON APRIL 17, 1998	STEVEN WAS BORN ON APRIL 17, 1998	Yes
..|-|.----.|...| -.|..|-.-.|.| -|---| --|.|.|-| -.--|---|..-|	IT'S NICE TO MEET YOU	IT'S NICE TO MEET YOU	Yes
....|---|.--| .-|.-.|.| -.--|---|..-| -..|---|..|-.|--.|..--..|	HOW ARE YOU DOING?	HOW ARE YOU DOING?	Yes
Inputting a sequence of more than 500 characters	Error message	Error message	Yes

3.2 Exercise 2 Tests

Table 2: Tests for exercise 2.

Description	**Expected**	**Result**	**Success**
Lora and Ben like emojis	lORA AND bEN LIKE EMOJIS	lORA AND bEN LIKE EMOJIS	Yes
abc	ABC	ABC	Yes
abcD	ABCd	ABCd	Yes
`\r#\r`	Number of bytes processed: 27, Number of Rx interrupts: 29, Number of Tx interrupts: 32	Number of bytes processed: 27, Number of Rx interrupts: 29, Number of Tx interrupts: 32	Yes

`\r%\r`	Byte Counter, CountRxIrq and CountTxIrq set to zero, Number of bytes processed: 0, Number of Rx interrupts: 0, Number of Tx interrupts: 0	CountTxIrq set to zero, Number of bytes processed: 0, Number of Rx interrupts: 0, Number of Tx interrupts: 0	Yes

4 Questions

1. Critical sections prevent race conditions among processes. It helps synchronize cooperative processes. In our code, the critical sections are used to prevent race conditions when reading/writing from global variables and queues. This ensures that when a global variable or queue is being modified or read from, nothing else will corrupt it or overwrite it while it is being read.

2. We do this to abstract away the implementation details so that we can focus on the main task at hand. The user task does not need to know about enabling and disabling interrupts in the receive and transmit directions.

3. The order does not matter. This is since the data is processed in the order it is receieved. It is therefore transmitted back in order too.

4. This prevents the interrupt from being triggered constantly due to the queue being empty.

5. The receive interrupts can be left on since they are not called if there is nothing to process.

6. When there aren't a lot of characters, the queue doesn't get full enough to send an interrupt; however, when a large number of characters are input, the queue gets full the the program uses the transmit queue instead. For example if we enter a small number of characters (ex. "abc") there are 6 of each number. However, if we enter more (ex. "abcabcabcabcabcabcabc") we get 24 bytes processed, 24 Rx interrupts, and 5 Tx interrupts.

5 Conclusion

In this lab, we

- gained experience using an ASCII-encoded serial communications connection between a PC host and a Zybo Z7 microcomputer board
- gained experience with the UART interface on the Zynq-7000 SoC
- gained experience interfacing using hardware and polling interrupts
- gained experience using queues to decouple the execution of software tasks

We successfully completed these objectives.

In exercise 1, we figured out how to implement a morse code decoder system using polled UART interfacing. We were able to successfully have our program interpret morse code and output text. This was done by having the terminal on the host PC send the morse code to the Zybo, having the Zybo decode the message into the alphanumeric message, and then having the Zybo send it back to the console.

In exercise 2, we successfully implemented an interrupt-driven driver for UART. We gained experience using the interrupt-driven method to transmit bytes to the SDK console through the UART. We also completed the four driver functions successfully in order to abstract implementation details away. We then completed the code in the main file to enable user input and output the correct results. We also implemented a way to clear the variables/counters.

By completeing these exercise, we gained experience with serial communication between the host and the Zybo, gained experience using the UART interface, gained experience with interrupts, and gained experience using queues.

6 Appendix

6.1 Exercise 1 Source Code

```
#include "xparameters.h"
#include "xplatform_info.h"
#include "xuartps.h"
#include "xil_exception.h"
#include "xil_printf.h"
#include "xscugic.h"

/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

#include "stdio.h"
#include "string.h"

// header file containing the morse code for different characters
#include "morseTranslator.h"

/************************** Constant Definitions *****************************/

//UART definitions from xparameters.h
#define UART_DEVICE_ID              XPAR_XUARTPS_0_DEVICE_ID
#define UART_BASEADDR                XPAR_XUARTPS_0_BASEADDR

//definitions for the terminating sequence characters
#define CHAR_HASH                0x23
#define CHAR_CARRIAGE_RETURN    0x0D

#define TASK_PROCESS_BUFFER    500
#define RECEIVER_POLL_PERIOD_MS 20

//queue limit definitions
#define XQUEUE_12_CAPACITY        20
#define XQUEUE_23_CAPACITY        30
//error message
#define ERROR_MESSAGE "Maximum message length exceeded. \
Message ignored.\rType in the termination sequence to \
translate the 'excess' characters that went above the array \
limit and caused overflow.\r\r\n"
```

```
/***************** Macros (Inline Functions) Definitions ********************/

static void TaskMorseMsgReceiver(void* pvParameters);

static TaskHandle_t xTask_msgreceive;

static void TaskMorseMsgProcessor(void* pvParameters);

static TaskHandle_t xTask_msgprocess;

static void TaskDecodedMsgTransmitter(void* pvParameters);

static TaskHandle_t xTask_msgtransmit;

static void UART_print_queueerror_msg(char* str);

static QueueHandle_t xQueue_12 = NULL;
static QueueHandle_t xQueue_23 = NULL;

void morseToTextConverter(char input_char);

/************************** Function Prototypes *****************************/

int Intialize_UART(u16 DeviceId);

/************************** Variable Definitions ****************************/

XUartPs Uart_PS;
XUartPs_Config* Config;

char char_morse_sequence[10];
char output_text_sequence[500];
int output_length;
int char_seq_length;

int main() {
    int Status;

    xTaskCreate(TaskMorseMsgReceiver,
                (const char*) "T1",
                configMINIMAL_STACK_SIZE * 5,
                NULL,
                tskIDLE_PRIORITY,
                &xTask_msgreceive);
```

```
    xTaskCreate(TaskMorseMsgProcessor,
                (const char*) "T2",
                configMINIMAL_STACK_SIZE * 5,
                NULL,
                tskIDLE_PRIORITY,
                &xTask_msgprocess);

    xTaskCreate(TaskDecodedMsgTransmitter,
                (const char*) "T3",
                configMINIMAL_STACK_SIZE * 5,
                NULL,
                tskIDLE_PRIORITY,
                &xTask_msgtransmit);


    Status = Intialize_UART(UART_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        xil_printf("UART Polled Mode Example Test Failed\r\n");
    }

    xQueue_12 = xQueueCreate(XQUEUE_12_CAPACITY, sizeof(u8));

    xQueue_23 = xQueueCreate(XQUEUE_23_CAPACITY, sizeof(u8));


    /* Check the queue was created. */
    configASSERT(xQueue_12);
    configASSERT(xQueue_23);

    xil_printf("Please type in your morse code below to be translated to text\n\n");
    xil_printf("Remember the following: \r        1) Add a '|' after every letter\r");
    xil_printf("        2) One space between each word\r");
    xil_printf("        3) One carriage return (ENTER) to write on a new line\r");
    xil_printf("        4) And when you are done, type in the termination sequence\n\n");

    vTaskStartScheduler();

    while (1);
    return 0;
}


int Intialize_UART(u16 DeviceId) {
    int Status;
```

```
    /*
     * Initialize the UART driver so that it's ready to use.
     * Look up the configuration in the config table, then initialize it.
     */
    Config = XUartPs_LookupConfig(DeviceId);
    if (NULL == Config) {
        return XST_FAILURE;
    }

    Status = XUartPs_CfgInitialize(&Uart_PS, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* Use NORMAL UART mode. */
    XUartPs_SetOperMode(&Uart_PS, XUARTPS_OPER_MODE_NORMAL);

    return XST_SUCCESS;
}

static void TaskMorseMsgReceiver(void* pvParameters) {
    u8 RecvChar;
    const TickType_t xPeriod = pdMS_TO_TICKS(RECEIVER_POLL_PERIOD_MS);
    TickType_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {
            vTaskDelayUntil(&xLastWakeTime, xPeriod);

        // Receive characters over UART if there's room in the queue
        if (uxQueueMessagesWaiting(xQueue_12) < XQUEUE_12_CAPACITY) {
                RecvChar = XUartPs_RecvByte(UART_BASEADDR);
            if (xQueueSendToBack(xQueue_12, &RecvChar, portMAX_DELAY) != pdPASS) {
                UART_print_queueerror_msg("xQueueSendToBack: Could not
                    write character to xQueue_12\n");
            }
        }
    }
}

static void TaskMorseMsgProcessor(void* pvParameters) {
    static char a[TASK_PROCESS_BUFFER];
    int no_of_characters_read;
    BaseType_t break_loop;
    int error_flag;
```

```
    for (;;) {
        no_of_characters_read = 0;
        break_loop = FALSE;
        output_length = 0;
        error_flag = 0;
        memset(output_text_sequence, 0, TASK_PROCESS_BUFFER);
            int sucess = pdFALSE;

            while (no_of_characters_read < TASK_PROCESS_BUFFER || sucess != pdTRUE) {
                no_of_characters_read++;
                sucess = xQueueReceive(xQueue_12,
                    &a[no_of_characters_read-1], portMAX_DELAY);
                        if (no_of_characters_read >= 4 &&
                                a[no_of_characters_read - 3] == CHAR_CARRIAGE_RETURN &&
                                a[no_of_characters_read - 2] == CHAR_HASH &&
                                a[no_of_characters_read-1] == CHAR_CARRIAGE_RETURN) {
                                for (int i = 0; i < no_of_characters_read; i++) {
                                        char c = a[i];
                                        morseToTextConverter(c);
                                }
                                break_loop = TRUE;
                                break;
                        }

        }

        if (!break_loop) {
                strcpy(a, ERROR_MESSAGE);
                        output_length = strlen(ERROR_MESSAGE);
                        error_flag = 1;
        }

        char* queue;
        if (error_flag == 1) {
            queue = a;
        } else {
            queue = output_text_sequence;
        }

                for (int i = 0; i < output_length; i++) {
                        xQueueSendToBack(xQueue_23, &queue[i], portMAX_DELAY);
                }
    }
}
```

```
static void TaskDecodedMsgTransmitter(void* pvParameters) {

    u8 write_to_console;
    const TickType_t xPeriod = pdMS_TO_TICKS(RECEIVER_POLL_PERIOD_MS);
    TickType_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {
        while (xQueueReceive(xQueue_23, &write_to_console,
            portMAX_DELAY) != pdTRUE);

        while (1) {
            if (XUartPs_IsTransmitFull(UART_BASEADDR)) {
                vTaskDelayUntil(&xLastWakeTime, xPeriod);
            } else {
                                XUartPs_SendByte(UART_BASEADDR, write_to_console);
                break;
            }
        }
    }
}

/*
 * Writes a null-terminated string to the UART device
 * set by UART_DEVICE_ID.
 */
static void UART_print_queueerror_msg(char* str) {
    int i = 0;
    while (TRUE) {
        XUartPs_SendByte(UART_BASEADDR, str[i]);
        ++i;

        // Do not print the null byte terminator
        if (str[i] == '\0')
            return;
    }
}
```

6.2 Exercise 2 Source Code

6.2.1 main

```
#include "stdio.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_types.h"
#include "xtime_l.h"

#include "uart_driver.h"
#include "initialization.h"

#define CHAR_ESC                0x23
#define CHAR_CARRIAGE_RETURN    0x0D
#define CHAR_PERCENT                    0x25

void printString(char countMessage[]);

TaskHandle_t task_receiveuarthandle = NULL;
TaskHandle_t task_transmituarthandle = NULL;
BaseType_t resetFlag;
BaseType_t endingSeq;
int Countbytes;
extern QueueHandle_t xQueue_for_transmit;
extern QueueHandle_t xQueue_for_receive;
extern int CountRxIrq;
extern int CountTxIrq;



//Function declaration for UART interrupt setup
extern int SetupInterruptSystem(INTC* IntcInstancePtr,
            XUartPs* UartInstancePtr, u16 UartIntrId);

//Initialization function for UART
extern int Initialize_UART();

//the four driver functions whose definitions are in "uart_driver.h" file.
extern BaseType_t MyIsReceiveData(void);

extern u8 MyReceiveByte(void);

extern BaseType_t MyIsTransmitFull(void);
```

```
extern void MySendByte(u8 Data);


int main() {
    int Status;

    xTaskCreate(Task_UART_buffer_receive, "uart_receive_task",
                1024, (void*) 0, tskIDLE_PRIORITY,
                &task_receiveuarthandle);
    xTaskCreate(Task_UART_buffer_send, "uart_transmit_task",
                1024, (void*) 0, tskIDLE_PRIORITY,
                &task_transmituarthandle);

    xQueue_for_transmit = xQueueCreate(SIZE_OF_QUEUE, sizeof(u8));
    xQueue_for_receive = xQueueCreate(SIZE_OF_QUEUE, sizeof(u8));
    CountRxIrq = 0;
    CountTxIrq = 0;
    Countbytes = 0;
    resetFlag = pdFALSE;
    endingSeq = pdFALSE;

    Status = Initialize_UART();
    if (Status != XST_SUCCESS) {
        xil_printf("UART Initialization failed\n");
    }

    vTaskStartScheduler();

    while (1);

    return 0;

}

void Task_UART_buffer_receive(void* p) {

    int Status;

    Status = SetupInterruptSystem(&InterruptController, &UART,
            UART_INT_IRQ_ID);
    if (Status != XST_SUCCESS) {
        xil_printf("UART PS interrupt failed\n");
    }
```

```
u8 returnFlag = 0;
u8 restartFlag = 0;

printString("Please enter the text and then press ENTER\n");
printString("To display the status messages,
        enter the end-of-block sequence, \\r#\\r and then
        press ENTER\n");
printString(
        "To reset the ISR transmit and receive global
        counters as well as byte counter, enter the
        sequence \\r%\\r and then press ENTER\n");

for (;;) {
    while (1) {
        u8 pcString;
        char write_to_queue_value;

        while (MyIsReceiveData() == pdFALSE) {};
                pcString = MyReceiveByte();
                while (MyIsTransmitFull() == pdTRUE) {};

        write_to_queue_value = (char) pcString;    //casted to "char" type.

        if (islower(write_to_queue_value)) {
                write_to_queue_value = toupper(write_to_queue_value);
        } else {
                write_to_queue_value = tolower(write_to_queue_value);
        }

        Countbytes++;
        MySendByte(write_to_queue_value);

        //detect \r#\r
        if (returnFlag == 2
                && write_to_queue_value == CHAR_CARRIAGE_RETURN) {
            returnFlag = 0;
            taskYIELD(); //force context switch
        } else if (returnFlag == 1 && write_to_queue_value == CHAR_ESC) {
            returnFlag = 2;
        } else if (write_to_queue_value == CHAR_CARRIAGE_RETURN) {
            returnFlag = 1;
        } else {
            returnFlag = 0;
        }
```

```
            if (restartFlag == 2 && write_to_queue_value == CHAR_CARRIAGE_RETURN) {
                    restartFlag = 0;
                CountRxIrq = 0;
                            CountTxIrq = 0;
                            Countbytes = 0;
                            xil_printf("\nByte Counter, CountRxIrq &&
                            CountTxIrq set to zero\n\n");
                taskYIELD(); //force context switch
            } else if (restartFlag == 1
                    && write_to_queue_value == CHAR_PERCENT) {
                    restartFlag = 2;
            } else if (write_to_queue_value == CHAR_CARRIAGE_RETURN) {
                    restartFlag = 1;
            } else {
                    restartFlag = 0;
            }
        }
    }
}


//print the provided number using driver functions
void printNumber(char number[]) {
    for (int i = 0; i < 10; i++) {
        if (number[i] >= '0' && number[i] <= '9') {
            while (MyIsTransmitFull() == pdTRUE);
            MySendByte(number[i]);
        }
    }
}

//print the provided string using driver functions
void printString(char countMessage[]) {
    for (int i = 0; countMessage[i] != '\0'; i++) {
        while (MyIsTransmitFull() == pdTRUE);
        MySendByte(countMessage[i]);
    }
}

void Task_UART_buffer_send(void* p) {

    UBaseType_t uxPriority;

    for (;;) {
        uxPriority = uxTaskPriorityGet(NULL);
```

```
        while (1) {

            //print the special messages for ending sequence
            char countArray[10];
            char CountRxIrqArray[10];
            char CountTxIrqArray[10];
            sprintf(countArray, "%d", Countbytes);
            sprintf(CountRxIrqArray, "%d", CountRxIrq);
            sprintf(CountTxIrqArray, "%d", CountTxIrq);
            printString("Number of bytes processed: ");
            printNumber(countArray);
            MySendByte(CHAR_CARRIAGE_RETURN);
            printString("Number of Rx interrupts: ");
            printNumber(CountRxIrqArray);
            MySendByte(CHAR_CARRIAGE_RETURN);
            printString("Number of Tx interrupts: ");
            printNumber(CountTxIrqArray);
            MySendByte(CHAR_CARRIAGE_RETURN);


            //perform context switch
            taskYIELD();
        }
    }
}
```

6.2.2 uart driver

```
#ifndef SRC_UART_DRIVER_H_
#define SRC_UART_DRIVER_H_

#include "xil_io.h"
#include "xuartps.h"
#include "xscugic.h"

/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "initialization.h"
```

```
void Task_UART_buffer_receive(void *p);

void Task_UART_buffer_send(void *p);

QueueHandle_t xQueue_for_transmit;
QueueHandle_t xQueue_for_receive;
int CountRxIrq;
int CountTxIrq;

#define UART_BASEADDR       XPAR_XUARTPS_0_BASEADDR

void Interrupt_Handler(void *CallBackRef, u32 Event, unsigned int EventData) {
    u8 receive_buffer;
    u8 transmit_data;
    u32 mask;
    if (Event == XUARTPS_EVENT_RECV_DATA) {
        BaseType_t xHigherPriorityTaskWoken;
        xHigherPriorityTaskWoken = pdFALSE;

        CountRxIrq++;

        while (XUartPs_IsReceiveData(UART_BASEADDR)) {
            receive_buffer = XUartPs_RecvByte(UART_BASEADDR);
            xQueueSendToBackFromISR(xQueue_for_receive,
                                    &receive_buffer, &xHigherPriorityTaskWoken);
        }
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    } else if (Event == XUARTPS_EVENT_SENT_DATA) {
        BaseType_t xHigherPriorityTaskWoken;
        xHigherPriorityTaskWoken = pdFALSE;
        CountTxIrq++;
        while (uxQueueMessagesWaitingFromISR(xQueue_for_transmit) > 0 &&
               !XUartPs_IsTransmitFull(XPAR_XUARTPS_0_BASEADDR)) {
            xQueueReceiveFromISR(xQueue_for_transmit,
                                    &transmit_data, &xHigherPriorityTaskWoken);
            XUartPs_SendByte(UART_BASEADDR, transmit_data);

        }
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        if (uxQueueMessagesWaitingFromISR(xQueue_for_transmit) <= 0) {
            mask = XUartPs_GetInterruptMask(&UART);
            XUartPs_SetInterruptMask(&UART, mask & ~XUARTPS_IXR_TXEMPTY);
        }
    } else {
        xil_printf("Nor a RECEIVE event nor a SEND event\n");
```

```
    }
}

BaseType_t MyIsReceiveData(void) {
    if (uxQueueMessagesWaiting(xQueue_for_receive) > 0) {
        return pdTRUE;
    } else {
        return pdFALSE;
    }
}

u8 MyReceiveByte(void) {
    u8 recv;
    taskENTER_CRITICAL();
    xQueueReceive(xQueue_for_receive, &recv, portMAX_DELAY);
    taskEXIT_CRITICAL();
    return recv;
}

BaseType_t MyIsTransmitFull(void) {
    if (uxQueueSpacesAvailable(xQueue_for_transmit) == 0) {
        return pdTRUE;
    } else {
        return pdFALSE;
    }
}

void MySendByte(u8 Data) {
    u32 mask;
    //secure it
    taskENTER_CRITICAL();

    mask = XUartPs_GetInterruptMask(&UART);
    XUartPs_SetInterruptMask(&UART, mask | XUARTPS_IXR_TXEMPTY);

    // if transmit FIFO empty, use polling,
    // otherwise insert to queue for interrupt method
    if (XUartPs_IsTransmitEmpty(&UART)
        && uxQueueMessagesWaiting(xQueue_for_transmit) == 0) {
        XUartPs_WriteReg(XPAR_XUARTPS_0_BASEADDR, XUARTPS_FIFO_OFFSET, Data);
    } else if (xQueueSend(xQueue_for_transmit, &Data, 0UL) != pdPASS) {
        xil_printf("Fail to send the data\n");
    }
    taskEXIT_CRITICAL();
}
```

```
#endif /* SRC_UART_DRIVER_H_ */
```