Lab 2

Polling and Interrupts

Michael Kwok

ECE 315 Lab H41
Department of Electrical and Computer Engineering
University of Alberta
24 March 2021

Contents

1 Abstract

The Zybo Z7 is a digital logic and embedded software development platform by Digilent, containing a Zybo 7000 System on a Chip (SoC) that has both a digital logic fabric (Xilinx 7-series FPGA) and a hard processor (ARM Cortex-A9). In this lab, we are interested in making use of the board's UART capabilities. The programs written for this lab will be running on FreeRTOS.

In the first part of the lab, we wrote 3 tasks to handle user input into a UART terminal, inverting the case of the text then sending it back via UART. In the second part, we were to write a task, helper functions and an Interrupt Service Routine to echo the text that the users input back through UART.

All code for this lab has been attached in the Appendix.

2 Design

2.1 Part 1

Three tasks and two queues were used in this program. This part of the lab relied purely on the FreeRTOS preemptive scheduler, as no task priority switching is done. This is done by having busy loops polling on data receive actions, indicating to the operating system that the task can get preempted out of execution.

In the first task, a busy loop just checks if the message queue between task one and two is full, and if it is not, it will read a byte from UART, sending the read value into the queue. This task can also get blocked in the loop to ensures that it only runs every 20 ms by using `vTaskDelayUntil`.

The second task checks the queue in an infinite loop until new data is available. When new data arrives, ut gets stored to a buffer local to the task. This task also checks input value order for the specified termination pattern, which was defined to be `\r#\r`. When the pattern is received, the task starts sending characters to the final task, converting the case of the characters as it does so, if it's an alphanumeric character.

In the final task, the characters are sent back to the UART port one character at a time with `XUartPs_Send` until the queue is empty.

2.2 Part 2

In the second part, only a single task is used as a lot of the work is done via an Interrupt Service Routine. The ISR first handles when data is received in the UART, sending the values read into a queue and incrementing a counter. When data is to be sent to the UART, the ISR checks if the FIFO is full before writing the values into the register. After everything has been written, the ISR masks the TXEMPTY interrupt to ensure that it does not get called multiple times on an empty queue.

Other functions in the `uart_driver.h` were written to replace the need to do polling in the main task. The functions are the following: `MyIsReceiveData()` to check if the interrupt handler has received any new data, `MyReceiveByte()` to read the received data,

`MyIsTransmitFull()` to check if the transmit queue is full and finally `MySendByte()` to send new data to the UART.

3 Testing

The first part of the lab was tested by putting in random text and symbols up to 64 characters long. Any more and the UART's FIFO buffer will get overwhelmed and will not get processed by the task. When a newline, a hash then another newline is entered, the processed text should show up in the serial monitor. Alphanumeric characters should have their case inverted, but symbols should stay the same.

For the 2nd part of the lab, as much text as possible can be put into the system, and it should get printed out back to the user after pressing enter. The user can check interrupt statistics by typing in a newline, a hash then another newline. To reset the statistics, a percent sign can be used instead of the hash.

4 Conclusion

Using interrupts has many advantageous over using polling. Only one task is required to handle user input, and much more data can be handled at one time as the input is processed as fast as the CPU can handle, while with polling there is extra waiting time involved.

There is a certain level of care that must be put into writing the interrupt service routine as it is critical to the performance of the program. An interrupt service routine doing too much work might slow everything down as it gets called very often.

5 Questions

- The critical sections are used to protect any form of globals and whenever there are queues being modified. This is to ensure that as global state is being modified, nothing can come in to corrupt it. In `MySendByte`, interrupt masks are being modified, and the queue is being is read from.
- The point of the driver is to add an abstraction over the implementation details. The user task should not know that an interrupt is being used to handle the input/output in the background.
- The order does not matter, as we're only trying to echo back what the user sent. The data is processed at the order it was received, so only the processed data will be output back.
- This is to ensure that the interrupt doesn't get constantly called due to the FIFO queue being empty.
- The interrupts can be left on, as they are not called if there is nothing to do.

- With lower input count, the FIFO queue don't get full enough to use the send interrupt. When a large amount of data is input at a time, the FIFO queue will get full, forcing the system to use the `xQueueTx` queue to supplement it instead.

A Part 1

```c
/*
 * ECE - 315 : WINTER 2021
 * LAB 2: Implementation of UART in polled mode - PART 1
 * Created by: Shyama M. Gandhi
 */

/***************************** Include Files *********************************/
#include <stdlib.h>

#include "xil_exception.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xplatform_info.h"
#include "xscugic.h"
#include "xuartps.h"
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
/************************** Constant Definitions *****************************/

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

#define CHAR_ESC 0x23 /* '#' character is used as termination sequence */
#define CHAR_CARRIAGE_RETURN \
  0x0D /* '\r' character is used in the termination sequence */

/***************************** Type Definitions ******************************/

/***************** Macros (Inline Functions) Definitions *********************/
static void TaskMsgReceiver(void *pvParameters);
static TaskHandle_t xTask_msgreceive;
static void TaskMsgProcessor(void *pvParameters);
static TaskHandle_t xTask_msgprocess;
static void TaskMsgTransmitter(void *pvParameters);
static TaskHandle_t xTask_msgtransmit;

static QueueHandle_t xQueue_12 = NULL;  // queue between task1 and task2
static QueueHandle_t xQueue_23 = NULL;  // queue between task2 and task3
/************************** Function Prototypes ******************************/

int Intialize_UART(u16 DeviceId);  // Initialization function for UART

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

XUartPs Uart_PS;        /* Instance of the UART Device */
XUartPs_Config *Config; /* The instance of the UART-PS Config */

int main(void) {
  int Status;

  xTaskCreate(TaskMsgReceiver,    /* The function that implements the task. */
              (const char *)"T1", /* 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, /* The task runs at the idle priority. */
                &xTask_msgreceive);

    xTaskCreate(TaskMsgProcessor, (const char *)"T2", configMINIMAL_STACK_SIZE,
                NULL, tskIDLE_PRIORITY, &xTask_msgprocess);

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

    // Initialization function for UART
    Status = Intialize_UART(UART_DEVICE_ID);
    if (Status != XST_SUCCESS) {
      xil_printf("UART Polled Mode Initialization Failed\r\n");
    }

    xQueue_12 = xQueueCreate(20, sizeof(u8));

    xQueue_23 = xQueueCreate(30, sizeof(u8));

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

    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 TaskMsgReceiver(void *pvParameters) {
    TickType_t xLastWakeTime;
```

```
  const TickType_t xFrequency = pdMS_TO_TICKS(20);
  for (;;) {
    u32 Running;
    u8 RecvChar;

    while (TRUE) {
        vTaskDelayUntil( &xLastWakeTime, xFrequency );
        if(uxQueueMessagesWaiting(xQueue_12) < 20){
            RecvChar = XUartPs_RecvByte(UART_BASEADDR);
            xQueueSend(xQueue_12, (void *)&RecvChar, portMAX_DELAY);
        }
    }
  }
}

static void TaskMsgProcessor(void *pvParameters) {
  u8 *read = malloc(1000 * sizeof(u8));
  int i = 0;

  for (;;) {
    u8 received_char;
    if (xQueueReceive(xQueue_12, (void *) &received_char, portMAX_DELAY) == pdPASS) {
      if (i < 1000) {
        read[i++] = received_char;

        if (i >= 4 && read[i - 3] == '\r' && read[i - 2] == '#' &&
            read[i - 1] == '\r') {
          for (int j = 0; j < i; j++) {
            u8 value = read[j];
            if (isalpha(value)) {
              value ^= 0x20;
            }
            xQueueSend(xQueue_23, (void *)&value, portMAX_DELAY);
          }
          i = 0;
        }
      } else {
        xil_printf("Maximum message length exceeded. Message ignored.\r\n");
        i = 0;
      }
    }
  }
  free(read);
}

static void TaskMsgTransmitter(void *pvParameters) {
  for (;;) {
    u8 write_to_console;

    while (uxQueueMessagesWaiting(xQueue_23) != 0) {
      xQueueReceive(xQueue_23, (void *)&write_to_console, 0);
      XUartPs_Send(&Uart_PS, &write_to_console, 1);
    }
  }
}
```

B Part 2

B.1 Driver

```
/*
 * uart_driver.h
 *
 *  Created on: Feb 28, 2021
 *      Author: Shyama M. Gandhi
 *
 *      Students are to add the driver code that has their definitions functions
 * in this file and use them to implement the interrupt method for reception as
 * well as transmission side. The current template uses interrupt method for
 * receiving and polling method for transmitting the data on the UART. Use the
 * lab manual instructions to implement the required interrupt method for
 * reception as well as transmission on UART. Please make sure that you use the
 * newly created functions to achieve it.
 *
 *      When using the idea to set threshold in Receive buffer, hint: have a
 * look at the function, XUartPs_SetFifoThreshold(UartInstancePtr,
 * UART_RX_BUFFER_SIZE);
 *
 */

#ifndef SRC_UART_DRIVER_H_
#define SRC_UART_DRIVER_H_

#include "xil_io.h"
#include "xscugic.h"  //interrupt controller header file
#include "xuartps.h"  //UART definitions header file

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

// UART Interrupt definitions
#define INTC XScuGic                            // Interrupt controller
#define UART_DEVICE_ID XPAR_XUARTPS_0_DEVICE_ID  // UART Device ID
#define INTC_DEVICE_ID \
  XPAR_SCUGIC_SINGLE_DEVICE_ID               // Interrupt controller device ID
#define UART_INT_IRQ_ID XPAR_XUARTPS_1_INTR  // UART interrupt identifier
#define UART_RX_BUFFER_SIZE 3U               // 5 bytes from host
#define SIZE_OF_QUEUE 100

void Task_UART_buffer_receive(void *p);
void Task_UART_buffer_send(void *p);

// UART interrupt control ISR declaration
void Interrupt_Handler(void *CallBackRef, u32 Event, unsigned int EventData);

XUartPs UART;              // UART Instance
XUartPs_Config *Config;    // Pointer to UART
INTC InterruptController;  // Interrupt controller instance
u32 IntrMask;  // interrupt mask variable to be used to enable different type of
               // interrupt on Rx and Tx in UART
```

```

QueueHandle_t xQueueRx;
QueueHandle_t xQueueTx;

int interrupt_counter = 0;
int CountRxIrq = 0;
int CountTxIrq = 0;

int Initialize_UART() {
  int Status;

  Config = XUartPs_LookupConfig(UART_DEVICE_ID);
  if (NULL == Config) {
    return XST_FAILURE;
    xil_printf("UART PS Config failed\n");
  }

  // Initialize UART
  Status = XUartPs_CfgInitialize(&UART, Config, Config->BaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
    xil_printf("UART PS init failed\n");
  }
  ResetStats();

  return XST_SUCCESS;
}

// Function for interrupt setup
int SetupInterruptSystem(INTC *IntcInstancePtr, XUartPs *UartInstancePtr,
                         u16 UartIntrId) {
  int Status;
  XScuGic_Config *IntcConfig;  // Config pointer for interrupt controller

  // Lookup the config information for interrupt controller
  IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
  if (NULL == IntcConfig) {
    return XST_FAILURE;
  }

  // Initialize interrupt controller
  Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                                 IntcConfig->CpuBaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  // Connect the interrupt controller interrupt handler
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                               (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                               IntcInstancePtr);

  // Connect the PS UART interrupt handler
  // The interrupt handler whichhandles the interrupts for the UART peripheral
  // is connected to it's unique ID number (82 in this case)
  Status = XScuGic_Connect(IntcInstancePtr, UartIntrId,
                           (Xil_ExceptionHandler)XUartPs_InterruptHandler,
                           (void *)UartInstancePtr);
```

```
112     if (Status != XST_SUCCESS) {
113       return XST_FAILURE;
114     }
115
116     // Enable the UART interrupt input on the interrupt controller
117     XScuGic_Enable(IntcInstancePtr, UartIntrId);
118
119     // Enable the processor interrupt handling on the ARM processor
120     Xil_ExceptionEnable();
121
122     // Setup the UART Interrupt handler function
123     XUartPs_SetHandler(UartInstancePtr, (XUartPs_Handler)Interrupt_Handler,
124                        UartInstancePtr);
125
126     // Create mask for UART interrupt, Enable the interrupt when the receive
127     // buffer has reached a particular threshold
128     IntrMask = XUARTPS_IXR_TOUT | XUARTPS_IXR_PARITY | XUARTPS_IXR_FRAMING |
129                XUARTPS_IXR_OVER | XUARTPS_IXR_TXEMPTY | XUARTPS_IXR_RXFULL |
130                XUARTPS_IXR_RXOVR;
131
132     // Setup the UART interrupt Mask
133     XUartPs_SetInterruptMask(UartInstancePtr, IntrMask);
134
135     // Setup the PS UART to Work in Normal Mode
136     XUartPs_SetOperMode(UartInstancePtr, XUARTPS_OPER_MODE_NORMAL);
137
138     return XST_SUCCESS;
139   }
140
141   // PS UART Interrupt Subroutine
142   void Interrupt_Handler(void *CallBackRef, u32 Event, unsigned int EventData) {
143     u8 data;
144     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
145     UBaseType_t uxSavedInterruptStatus;
146
147     uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();
148     if (Event == XUARTPS_EVENT_RECV_DATA) {
149       while (XUartPs_IsReceiveData(XPAR_XUARTPS_0_BASEADDR)) {
150         data = XUartPs_ReadReg(XPAR_XUARTPS_0_BASEADDR, XUARTPS_FIFO_OFFSET);
151         xQueueSendFromISR(xQueueRx, (void *)&data, &xHigherPriorityTaskWoken);
152         interrupt_counter++;
153       }
154       CountRxIrq++;
155     } else if (Event == XUARTPS_EVENT_SENT_DATA) {
156       while (!XUartPs_IsTransmitFull(XPAR_XUARTPS_0_BASEADDR)) {
157         if (xQueueReceiveFromISR(xQueueTx, &data, &xHigherPriorityTaskWoken)) {
158           XUartPs_WriteReg(XPAR_XUARTPS_0_BASEADDR, XUARTPS_FIFO_OFFSET, data);
159           interrupt_counter++;
160         } else {
161           XUartPs_SetInterruptMask(
162               &UART, XUartPs_GetInterruptMask(&UART) & ~XUARTPS_IXR_TXEMPTY);
163           break;
164         }
165       }
166       CountTxIrq++;
167     }
168     taskEXIT_CRITICAL_FROM_ISR(uxSavedInterruptStatus);
169
```

```
170     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
171   }
172
173   void ResetStats() {
174     interrupt_counter = 0;
175     CountTxIrq = 0;
176     CountRxIrq = 0;
177   }
178
179   void SendStats() {
180     xil_printf("Number of bytes processed: %d\n", interrupt_counter);
181     xil_printf("Number of Rx interrupts: %d\n", CountTxIrq);
182     xil_printf("Number of Tx interrupts: %d\n", CountRxIrq);
183   }
184
185   BaseType_t MyIsReceiveData() { return uxQueueMessagesWaiting(xQueueRx) != 0; }
186
187   u8 MyReceiveByte() {
188     u8 data;
189     xQueueReceive(xQueueRx, (void *)&data, 0);
190
191     return data;
192   }
193
194   BaseType_t MyIsTransmitFull() {
195     return uxQueueMessagesWaiting(xQueueTx) == SIZE_OF_QUEUE;
196   }
197
198   void MySendByte(u8 data) {
199     taskENTER_CRITICAL();
200
201     XUartPs_SetInterruptMask(
202         &UART, XUartPs_GetInterruptMask(&UART) | XUARTPS_IXR_TXEMPTY);
203
204     if (XUartPs_IsTransmitFull(XPAR_XUARTPS_0_BASEADDR)) {
205       xQueueSend(xQueueTx, (void *)&data, 0);
206     } else {
207       XUartPs_WriteReg(XPAR_XUARTPS_0_BASEADDR, XUARTPS_FIFO_OFFSET, data);
208     }
209
210     taskEXIT_CRITICAL();
211   }
212
213   #endif /* SRC_UART_DRIVER_H_ */
```

B.2 Main

```
/*
 * uart_driver.h
 *
 *  Created on: Feb 28, 2021
 *      Author: Shyama M. Gandhi
 *
 *     This is the main file that uses the Xilinx the "uart_driver.h" that you
 * will modify in this lab. Receive side: implemented in interrupt mode Transmit
 * side: implemented in polling mode Hints about the places where you will find
 * info for creating four drivers functions and using them:
 *     1. xuartps_hw.h
 *     2. xuartps.h
 *     3. Use the lecture slides to get more idea about the UART functionality
 *     4. Introduction presentation
 *
 *     There are two tasks in this file. One task will block on the receive
 * queue and will pass any characters if present in the transmit queue. Another
 * task transfers bytes from the transmit queue into the TxFIFO and hence you
 * see the echoed characters back on the console based on what you typed.
 *
 *     Study the Interrupt Mask function XUartPs_SetInterruptMask() carefully.
 *     This functions has details that can be used to enable different types of
 * interrupts in Tx and Rx direction.
 *
 */

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

// this is the uart driver file where students will add the implementation as
// mentioned in the lab manual
#include "uart_driver.h"

TaskHandle_t task_receiveuarthandle = NULL;
TaskHandle_t task_transmituarthandle = NULL;

// Function declaration for UART interrupt setup
extern int SetupInterruptSystem(INTC *IntcInstancePtr, XUartPs *UartInstancePtr,
                                u16 UartIntrId);
// Initialization function for UART
extern int Initialize_UART();
extern void Transmit_Byte(u8 data);

extern QueueHandle_t xQueueRx;
extern QueueHandle_t xQueueTx;

// functions to be implemented by students
extern BaseType_t MyIsReceiveData();
extern u8 MyReceiveByte();
extern BaseType_t MyIsTransmitFull();
extern void MySendByte(u8 Data);
extern void ResetStats();
extern void SendStats();
```

```c

int main() {
  int Status;
  xTaskCreate(Task_UART_buffer_receive, "uart_receive_task", 1024, (void *)0,
              tskIDLE_PRIORITY, &task_receiveuarthandle);
  // Set up queues
  xQueueRx = xQueueCreate(SIZE_OF_QUEUE, sizeof(u8));
  xQueueTx = xQueueCreate(SIZE_OF_QUEUE, sizeof(u8));

  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 ringbuf[3];

  while (1) {
    while (!MyIsReceiveData())
      ;
    u8 data;
    data = MyReceiveByte();

    while (MyIsTransmitFull())
      ;
    MySendByte(data);

    ringbuf[2] = ringbuf[1];
    ringbuf[1] = ringbuf[0];
    ringbuf[0] = data;

    if (ringbuf[0] == '\r' && ringbuf[2] == '\r') {
      if (ringbuf[1] == '%') {
        ResetStats();
      } else if (ringbuf[1] == '#') {
        SendStats();
      }
    }
  }
}
```