# ECE 315 Lab 3

Lora Ma
Benjamin Kong

ECE 315 Lab Section H41

March 28, 2022

## Contents

## 1 Abstract

The purpose of this lab was to

- gain experience using the serial peripheral interface in both the master and slave modes
- gain experience creating artificial load on a CPU and then measuring the resulting load using `vTaskGetRunTimeStats()`.

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 be using CPU0 to run the FreeRTOS real-time kernel which will have three non-idle tasks for the SPI interface.

In this lab, we will be completing two exercises. Exercise 1 is to verify the given system and add the byte count message. The program will echo characters that are type on the keyboard until the termination sequence `\r#\r` is detected. Then, a string will be generated and sent over SPI which will look something like `The number of characters received over SPI: <number> \n` where `<number>` is the total bytes that were received over the SPI1 interface.

In exercise 2, we will be calibrating the load generator experimentally. The program prints the percentage values of time consumed by the tasks `TaskCPULoadGen`, `TaskLoopCountProcessor`, and `TaskPrintRunTimeStats`. It will be displayed in a tabular format on the terminal. We will the utilize this program to experimentally determine the value of `loop_count`.

## 2 Design

### 2.1 Exercise 1 Design

A systematic diagram that illustrates the relationship between the SDK terminal on the host, the UART interface, the SPI0 and SPI1 interfaces, and the three tasks is shown below.

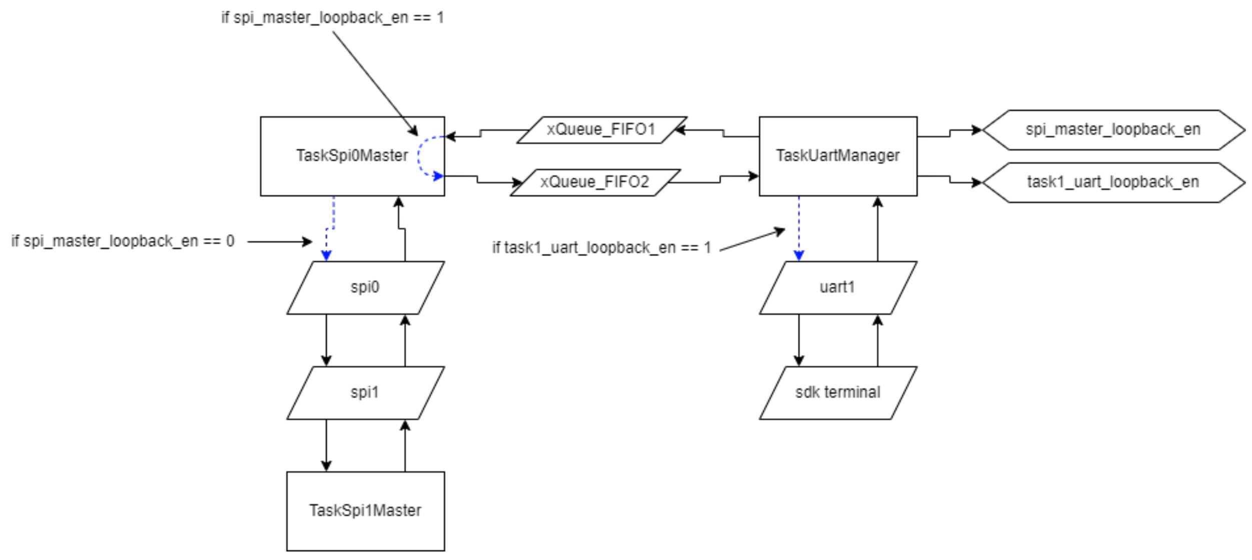

**Figure 1:** Systematic diagram showing the connections between each element.

Below, the block design for this exercise is shown. The confections between SPI0 and SPI1 are clearly visible. For example, `SPI0_MOSI_I` is connected to `SPI1_MOSI_O`, `SPI0_MOSI_O` is connected to `SPI1_MOSI_I`, `SPI0_MISO_I` is connected to `SPI1_MISO_O`, and `SPI0_MISI_O` is connected to `SPI1_MISO_I`.

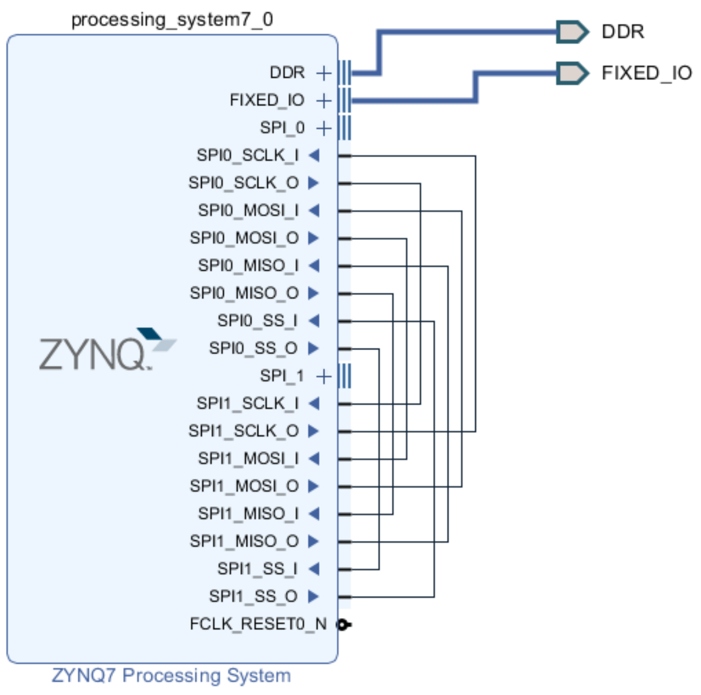

**Figure 2:** Block design for this exercise.

In exercise 1, we first completed the `TaskUartManager` function. We wrote the logic to send a dummy character using FIFO1 and then receive the bytes from FIFO2 once the `TaskSpi0Master` responds.

The second thing we completed was the `TaskSpi0Master` function. We first copy the received data from FIFO1 into a buffer. We transfer bytes based on the `TRANSFER_SIZE_IN_BYTES` value. For this lab, the value was set to 1 (i.e. 1 byte). We use the `bytecount` variable to keep track of how many bytes we have stored in the buffer and compare it to `TRANSFER_SIZE_IN_BYTES` to determine if we need to sent the data yet.

Once we have determined we need to transfer, we call `SpiMasterWrite` to transfer the bytes to the slave. We then call `taskYIELD()` to allow the slave to work. Once the slave yields back to this task, we read data from master into `send_SPI_data_via_FIFO2` (this variable is a variable to help us store intermediary results). We then send this to data to the back of FIFO2. Lastly, we reset `bytecount` back to 0 to allow data to accumulate in the buffer again before repeating this whole process again.

The last function we completed was the `TaskSpi1Slave` function. We wrote logic to detect the termination sequence `\r#\r`. This was done by incrementing the `end_sequence_flag` variable whenever the correct termination character appeared and resetting it to 0 otherwise. If the flag reached 3 (the length of `\r#\r`) then we proceed to the logic for handling termination.

Our goal when the termination sequence is triggered is to send the bytes that make up the message "The number of characters received over SPI:%d" to the master SPI. In order to do this, we set the `flag` variable to 1 and then used a for loop to send the bytes. This is done by using `SpiSlaveWrite` and sending 1 byte. We then use `SpiSlaveRead` to receive the response. Once all the bytes have been sent by using the for loop, we reset our counters/flags to allow this process to repeat.

## 2.2 Exercise 2 Design

In exercise 2, we attempted to find values of `loop_count` that result in 90%, 80%, 70%, 60%, 50%, and 40% idle load percentages. In order to do this, we completed the necessary functions and testing varying values of `loop_count` until we found the desired idle load percentages. The table below shows the values of `loop_count` that resulted in the aforementioned idle load percentages.

**Table 1:** `loop_count` values for the corresponding idle load percentages

| Load | 90% | 80% | 70% | 60% | 50% | 40% |
|---|---|---|---|---|---|---|
| Loop count | 50000 | 125000 | 200000 | 275000 | 350000 | 400000 |

For `TaskLoopCountProcessor`, we set the `delay` variable to 90000 ms if `loop_count` exceeded 500000 and 120000 ms if `loop_count` exceeded 1000000. We also incremented the `loop_count` variable by 25000 for each loop.

For the `TaskCpuLoadGen` function, we needed something to execute to simulate CPU load. To do this, we iterated `loop_count` number of times, each time taking the bitwise complement of a dummy variable. This was done via `var = !var`.

Lastly, for the `TaskPrintRunTimeStats` function, we needed to output the runtime stats using `vTaskGetRunTimeStats`. We first enabled this feature as described in the helper document. We then created a character buffer and wrote the runtime stats into the buffer using `vTaskGetRunTimeStats`. We then used `xil_printf` to output the buffer which contains the runtime stats.

## 3 Testing

### 3.1 Exercise 1 Tests

### 3.2 Exercise 2 Tests

## 4 Conclusion

The purpose of this lab was to

- gain experience using the serial peripheral interface in both the master and slave modes
- gain experience creating artificial load on a CPU and then measuring the resulting load using `vTaskGetRunTimeStats()`.

We believe we have fully completed the objectives of this lab.

In exercise 1, we gained experience using the SPI interface. The program echos characters that are typed on the keyboard until the termination sequence is detected. Then, it will generate a string to be sent over SPI where it will print the number of total bytes that were received over the SPI interface.

In exercise 2, we created a program that prints the percentage values of time consumed by three tasks. We summarized this information to be printed in a tabular format in the terminal. This program was then used to experimentally determine the value of `loop_count`.

We were able to gain experience using the serial peripheral interface (SPI) in both master and slave modes. We were also able to gain experience creating artificial load on a CPU and then measuring the resulting load using `vTaskGetRunTimeStats()`.

## 5 Appendix

### 5.1 Exercise 1 Source Code

The source code is in the file `spi_lab3_main.c` that was submitted along with this report.

### 5.2 Exercise 2 Source Code

The source code is in the file `load_gen_main.c` that was submitted along with this report.