

ECE 315

Lab Section H31

Lab 3

Due Date: April 9, 2022

Name:

- Chanpreet Singh
- Gurbani Baweja

Abstract

This lab was centred around using the Zynq-7000 SPI Interface to learn about interfacing using the serial peripheral interface (SPI). We completed two exercises - verifying a given system and adding a byte count message to the system, and calibrating a load generator experimentally. The objective of this lab was to acquire the knowledge of using serial peripheral interface (SPI) in both the modes i.e., - master (controller) and slave (peripheral). Another goal of this lab was to acquire the knowledge of creating an artificial load on the CPU. After the load was created artificially, our objective was to measure the resulting load using the FreeRTOS function - `vTaskGetRunTimeStats()`.

To accomplish our objectives from this lab, we used a Digilent Zybo Z7 development board as the hardware platform, however a Digilent Cora Z7 can also be used. CPU0 ran the FreeRTOS real-time kernel with three non-idle tasks for the SPI interface. We downloaded the fixed Zynq-7000 System-On-Chip (SoC) hardware configuration and the initial skeleton source file and modified the initial application code in C for implementing our designs for Exercise 1 and Exercise 2.

Exercise 1 of this lab was aimed at verifying the given system and adding a byte count message to the given system. The Open Block Design command in the IP INTEGRATOR submenu was executed and a diagram depicting the connections between the two SPI interfaces - SPI0 and SPI1 was obtained (This diagram is attached in the design section along with an explanation of the connections). We first verified the system provided to us by executing several commands like - Generate Bitstream in the PROGRAM AND DEBUG submenu and Run As -> Launch on Hardware (System Debugger) command to compile the project. We then verified whether the command Loopback TaskUartManager and Loopback TaskSpi0Master functions properly. Finally, we verified whether the default behaviour in TaskSpi1Slave works correctly and then modified the TaskSpi1Slave to add the byte count message. TaskSpi1Slave had to detect the message termination sequence (Enter, #, Enter) in the stream of characters and generate a string which was sent immediately after the received bytes were echoed over SPI back to TaskSpi0Master.

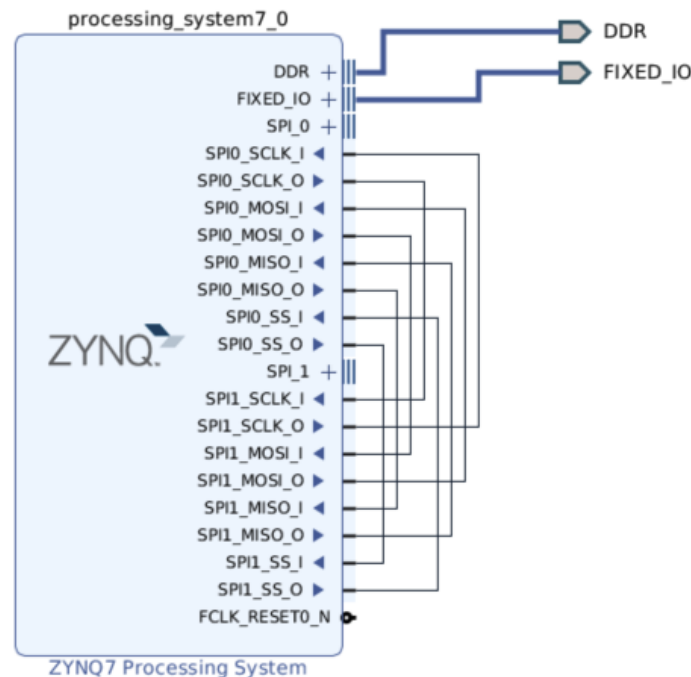
For adding the byte count message, when the message termination sequence (Enter, #, Enter) is detected by this task in the stream of characters, a string will be generated by TaskSpi1Slave that is sent immediately after the received bytes have been echoed over SPI back to TaskSpi0Master. The generated string was - ": The number of characters received over SPI: <number> \n ", where the total number of bytes that were received by TaskSpi1Slave over the SPI1 interface were depicted by <number> .

Exercise 2 of the lab was aimed at calibrating a load generator experimentally, where we modified the `load_gen_main.c` file, which generated an artificial CPU load controlled by global u32 variable `loop_count`. Some important modifications were made to the Xilinx SDK BSP folder following the instructions provided in the instruction document on eclass. Three tasks were modified to complete this exercise - TaskCpuLoadGen, TaskLoopCountProcessor, and TaskPrintRunTimeStats and finally, by trial and error experiments, we determined the `loop_count` values that correspond to IDLE time percentages of 40%, 50%, 60%, 70%, 80% and 90%.

Design:

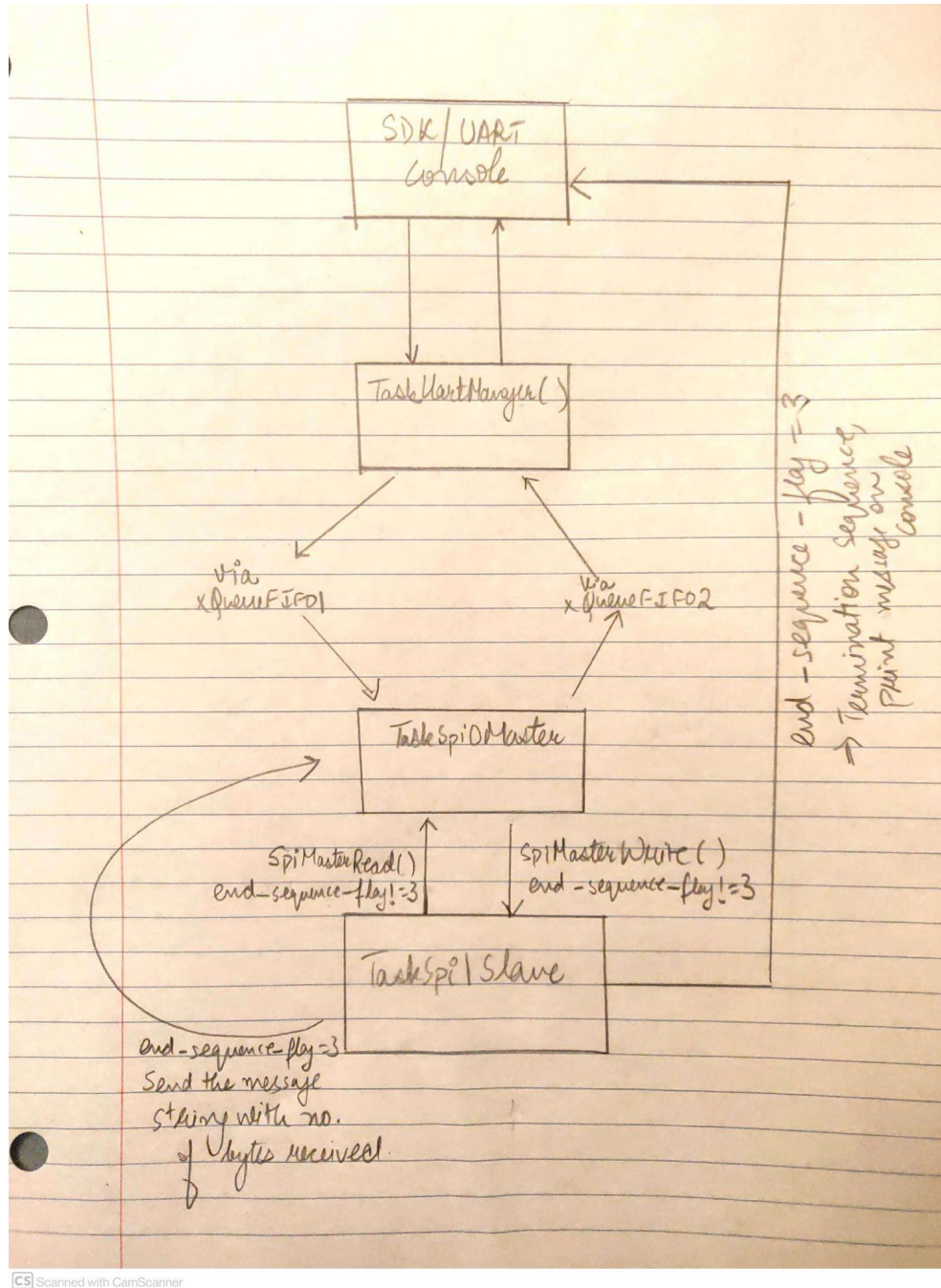
The design for Exercise 1 per task is explained as follows:

After executing the Open Block Design command in the IP INTEGRATOR submenu, we obtained the diagram as shown below:



The diagram above shows the connections between the two SPI interfaces - SPI0 and SPI1. As can be seen, the various input and output connections of SPI0 and SPI1 are connected to each other respectively. For instance - the SPI0_MOSI_I is connected to the SPI1_MOSI_O, SPI1_SCLK_O is connected to the SPI1_SCLK_I and others are connected in a similar fashion. Thus, the different input and output connections between input and output of SPI0 and SPI1 ensure that the SPI interface is working so that master and slave can communicate properly.

The diagram designed for the pre-lab is as follows:



In the task - TaskBar Manager, we sent the "dummy" control character using the FIFO1 xQueueSendToBack on the xQueue_FIFO1 and used the xQueueReceive(xQueue_FIFO2, &task1_receive_from_FIFO2_spi_data, portMAX_DELAY); function with the stated parameters to wait on to receive the bytes coming from the SPIMaster task via FIFO2. The XUartPs_IsTransmitFull function was used to determine whether there is space on the Transmitter UART side, and the XUartPs_SendByte was used send it to the UART.

In the task - TaskSpi0Master, the received data from the FIFO1 was copied into the "sendbuffer" variable using the SpiMasterWrite(send_buffer, TRANSFER_SIZE_IN_BYTES) and after that we incremented the bytecount and used task_YIELD() to allow the slave SPI task to work. Finally, we used SpiMasterRead to read from the master implementation and then sent the data to the back of the FIFO2 using xQueueSendToBack and reset bytecount to zero.

In the task - TaskSpi1Slave, we detected the termination sequence by using the end_sequence_flag. We sent the byte back to the SPI master until the termination sequence by checking that end_sequence_flag is not equal to three and using SpiSlaveRead to receive the bytes. The variable num_received was incremented as bytes were received. Then we used the Implementation of checkForTerminationSequence() for checking for the terminations sequence and sent the bytes to TaskSpi0Master using SpiSlaveWrite. Once the termination sequence was detected, we printed the string message using xil_printf() and sent the string to master using SpiSlaveWrite. Finally, we reset the values of end_sequence_flag and num_received to zero.

The design for Exercise 2 is explained as follows:

In this exercise, we calibrate the load generator experimentally to artificially generate CPU load with the global variable loop_count. We start by setting an initial value for loop_count, and then delay it for 90,000 if the loop_count is above 500,000 or we delay it for 120,000 if the loop_count is greater than 1000,000 in the TaskLoopCountProcessor(). We also increment loop_count by 5,000 after each loop cycle. and delay the task for 1 tick time. Then we create the fake CPU load in TaskCpuLoadGen() by executing a simply bitwise complement operation for loop_count number of times on a variable var and delay the task by 1 tick time. Finally, we print the statistics with IDLE task and CPU load, by using vTaskGetRunTimeStats().

Test Suit:

Test ID	Description	Expected Result	Actual Result	Success/Failure
T1	Enter command 1 to enable Task 1 Loop Back Mode	***UART Manager Task loopback enabled*** on the SDK Terminal	***UART Manager Task loopback enabled*** on the SDK Terminal	Success
T2	After command 1, enter Gurbani in the SDK Terminal	Gurbani displayed on the SDK Terminal	Gurbani displayed on the SDK Terminal	Success
T3	Enter command # to diasble Task 1 Loop	***UART Manager Task loopback	***UART Manager Task loopback	Success

	Back Mode from the enable mode	disabled using command toggling*** on the SDK Terminal	disabled using command toggling*** on the SDK Terminal	
T4	After disabling Loop Back mode, enter Hello in the SDK Terminal	Hello not echoed back on the SDK Terminal	Hello not echoed back on the SDK Terminal	Success
T5	Enter command 2 to enable Task 2 Loop Back Mode	***Task 2 loopback enabled : No access to the SPI0 interface*** on the SDK Terminal	***Task 2 loopback enabled : No access to the SPI0 interface*** on the SDK Terminal	Success
T6	After command 2, enter Gurbani in the SDK Terminal	Gurbani displayed on the SDK Terminal	Gurbani displayed on the SDK Terminal	Success
T7	Enter command 2 to disable Loop Back Mode and gain access to SPI, then enter Gurbani in the SDK Terminal	Gurbani displayed on the SDK Terminal	Gurbani displayed on the SDK Terminal	Success
T8	After implementing T7,, give termination sequence - #	The number of characters received over SPI:7 displayed on the SDK Terminal	The number of characters received over SPI:7 displayed on the SDK Terminal	Success

Note: While implementing T7 and T8, we received some garbage bytes on the SDK Terminal and not what we expected. The causes for the incorrect results are explained in the Conclusion segment.

The results obtained from exercise two are attached in the results segment.

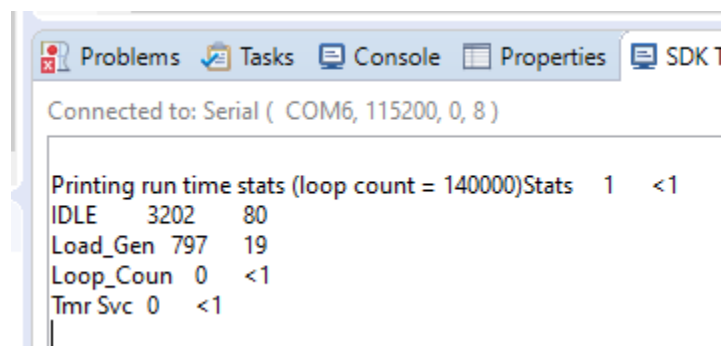
Result:

In exercise 1, we were able to successfully verify the given system and some of the test cases regarding that are shown in the test cases table and the addition of a byte count message was also implemented.

The results for exercise 2, i.e., the values of loop_count that correspond to IDLE time percentages of 40%, 50%, 60%, 70%, 80% and 90%. by trial-and-error experiments are as shown below:

IDLE Time Percentage	Value of loop_count
40%	450000
50%	400000
60%	350000
70%	250000
80%	140000
90%	74000

We can observe from the table above that as the IDLE Time Percentage increases, the value of the loop_count i.e., the load factor caused on the CPU decreases. This is because, the CPU utilization is the time the CPU is not running an idle thread, and so more the time CPU runs idle threads, i.e., more the IDLE Time Percentage, the less will be the loop_count i.e., the load factor caused on the CPU. A screenshot of the execution of exercise 2 for IDLE time percentage of 80% is as shown below:



```
Problems Tasks Console Properties SDK 1
Connected to: Serial ( COM6, 115200, 0, 8 )

Printing run time stats (loop count = 140000)Stats 1 <1
IDLE 3202 80
Load_Gen 797 19
Loop_Coun 0 <1
Tmr Svc 0 <1
```

Conclusion:

In this lab, we aimed at getting the knowledge of using the serial peripheral interface (SPI) in both the modes i.e., - master (controller) and slave (peripheral). Additionally, we aimed gaining the knowledge of creating an artificial load on the CPU artificially, and measured the resulting load using the FreeRTOS function - vTaskGetRunTimeStats(). We performed two exercises in this lab where the first exercise was aimed at verifying a given system and adding a byte count

message to the system and the second one was aimed at calibrating a load generator experimentally. A Digilent Zybo Z7 development board was used as the hardware platform.

In exercise one, we executed the Open Block Design command in the IP INTEGRATOR submenu to obtain a diagram depicting the connections between the two SPI interfaces - SPI0 and SPI1 was obtained (This diagram is attached in the design section along with an explanation of the connections). The system was verified using several commands and then the TaskSpi1Slave was modified to add the byte count message. A string (": The number of characters received over SPI: <number> \n ") was generated by TaskSpi1Slave when the message termination sequence (Enter, #, Enter) is detected by it in the stream of characters, and that string is sent immediately after the received bytes have been echoed over SPI back to TaskSpi0Master. We observed that the system was verified correctly by running the commands provided in the lab Handout and some of the test cases corresponding to the verification have been shown in the test cases table.

While at the disabled Loop Back Mode, accessing the SPI, we received garbage bytes on the SDK Terminal when we sent a message to the terminal and executed the termination command to get the number of bytes. As explained by the TA, this was because of the improper positioning of SpiMasterWrite(send_buffer, TRANSFER_SIZE_IN_BYTES) and taskYIELD() in the TaskSpi0Master task. These should have been inside the if statement where we checked if the bytecount is equal to TRANSFER_SIZE_IN_BYTES. We also incorrectly performed the temporary storage in the TaskSpi1Slave task. We placed temp_store = *RxBuffer_Slave before SpiSlaveRead(TRANSFER_SIZE_IN_BYTES) while it should have been written after SpiSlaveRead(TRANSFER_SIZE_IN_BYTES). These possible fixes can possibly solve the garbage problem at the SDK terminal.

In exercise two, we calibrated a load generator experimentally by modifying three tasks - TaskCpuLoadGen, TaskLoopCountProcessor, and TaskPrintRunTimeStats in the load_gen_main.c file. Some important modifications were also made to the Xilinx SDK BSP folder following the instructions provided in the instruction document on eclass. By trial and error experiments, we determined the loop_count values that correspond to IDLE time percentages of 40%, 50%, 60%, 70%, 80% and 90% which are shown in the table in the results segment. From the loop_count values obtained, we realized the relationship between loop_count and IDLE time percentages. So, we observed that as the IDLE Time Percentage increases, the value of the loop_count i.e., the load factor caused on the CPU decreases. Since more the time the CPU is running idle, less the load factor caused on the CPU will be, therefore, the loop_count decreases.