ECE 315: Computer Interfacing

H21 Lab 3 - Zynq-7000 SPI Interface

Timmy Ifidon
CCID: kifidon

Mekha George
CCID: mekha

Mar 25, 2025

Abstract:

This lab focuses on the implementation of SPI loopback communication on the Zybo Z7 FPGA and the development of a FreeRTOS-based embedded system modeled after *Space Invaders*. The primary objective was to gain a deeper understanding of SPI integration, data communication in both master and slave modes, and practical experience interfacing with SPI peripherals. The lab was divided into two parts: Part 1 explored SPI loopback communication between FreeRTOS tasks, emphasizing inter-task communication and synchronization. In Part 2, the Xilinx/AMD Pmod OLED module was used as a display for an embedded system simulating a *Space Invaders* game, incorporating a combination of polling and queued task communication to manage the display updates. This lab provided hands-on experience in system architecture design within FreeRTOS, real-time task management, and SPI-based peripheral integration.

Design:

Part 1

Part 1 of the lab focused on implementing and testing SPI loopback functionality. The objective was to enable communication between SPI master and slave interfaces, allowing data to be echoed back in a loopback mode. The system provided real-time feedback via the terminal and RGB LEDs, which indicate different loopback states. The final task involved modifying the SPI slave task to track received data and provide statistical summaries. The general flow of data is as follows:

1. UART Loopback Mode
    a. Initially, loopback is disabled.
    b. Enabling loopback will cause typed text to be echoed back immediately in the terminal.
    c. Commands are entered as \r<command>\r, and \r%\r terminates text entry.
    d. The RGB LED turns red to indicate UART loopback mode.
2. SPI Loopback Mode
    a. Initially, loopback is disabled.
    b. Enabling SPI loopback will cause text entered in the terminal to be echoed back via SPI.
    c. The RGB LED turns blue when SPI loopback is enabled and green when disabled.
3. SPI Subtask
    a. When SPI loopback is disabled, SPI communication with an external device should be enabled
    b. The SPI subtask processes received data to track and count received messages.
    c. Upon detecting \r%\r, a summary message showing the total bytes received and the number of messages processed is printed to the terminal.

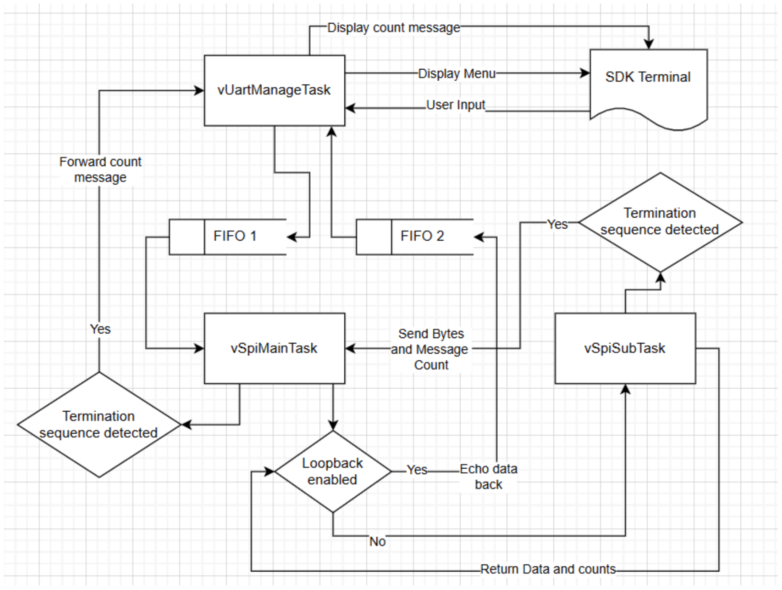


Figure 3.1.1 - Task Flow Diagram: The system supports UART and SPI loopback modes. Initially, both loopbacks are disabled. Enabling UART loopback causes typed text to be immediately echoed back in the terminal. Commands are entered as \r<command>\r, and \r%\r terminates the entry. Enabling SPI loopback echoes text back via SPI. When SPI loopback is disabled, external SPI communication is enabled, and the SPI subtask processes received data to track and count messages. Upon detecting \r%\r, a summary of total bytes and messages processed is printed.

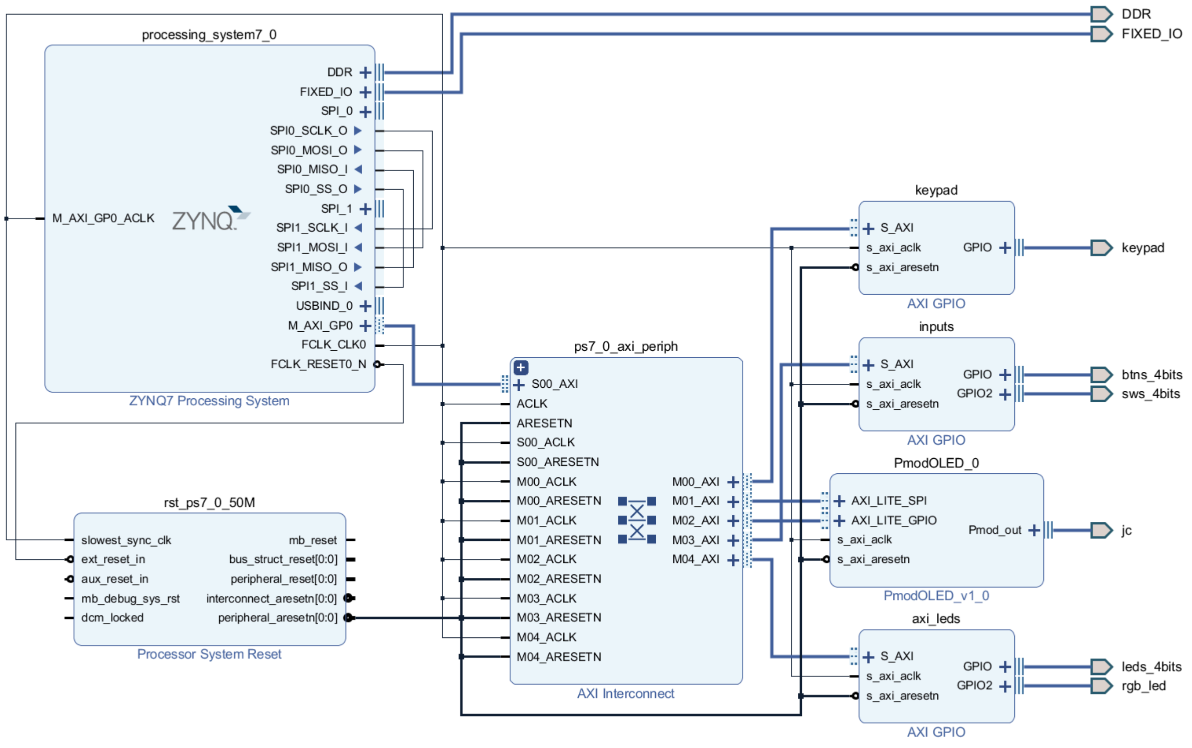
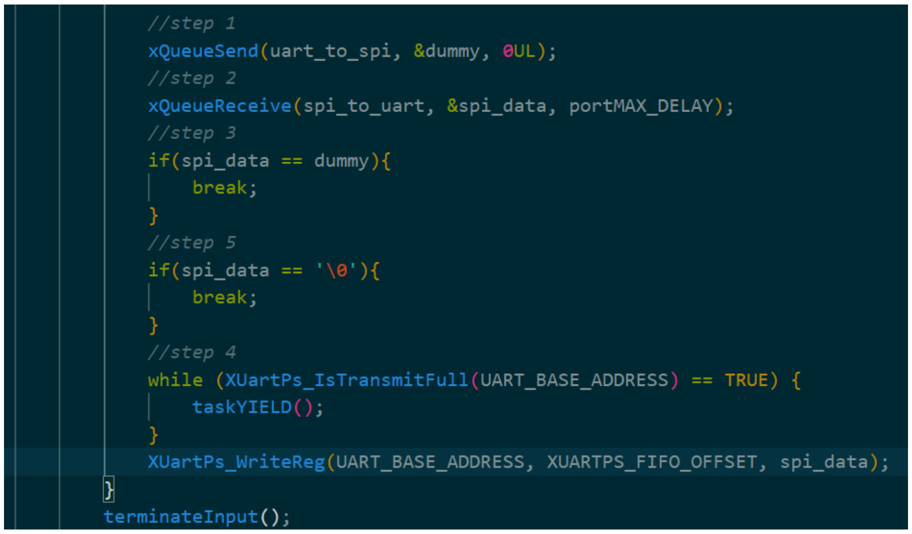
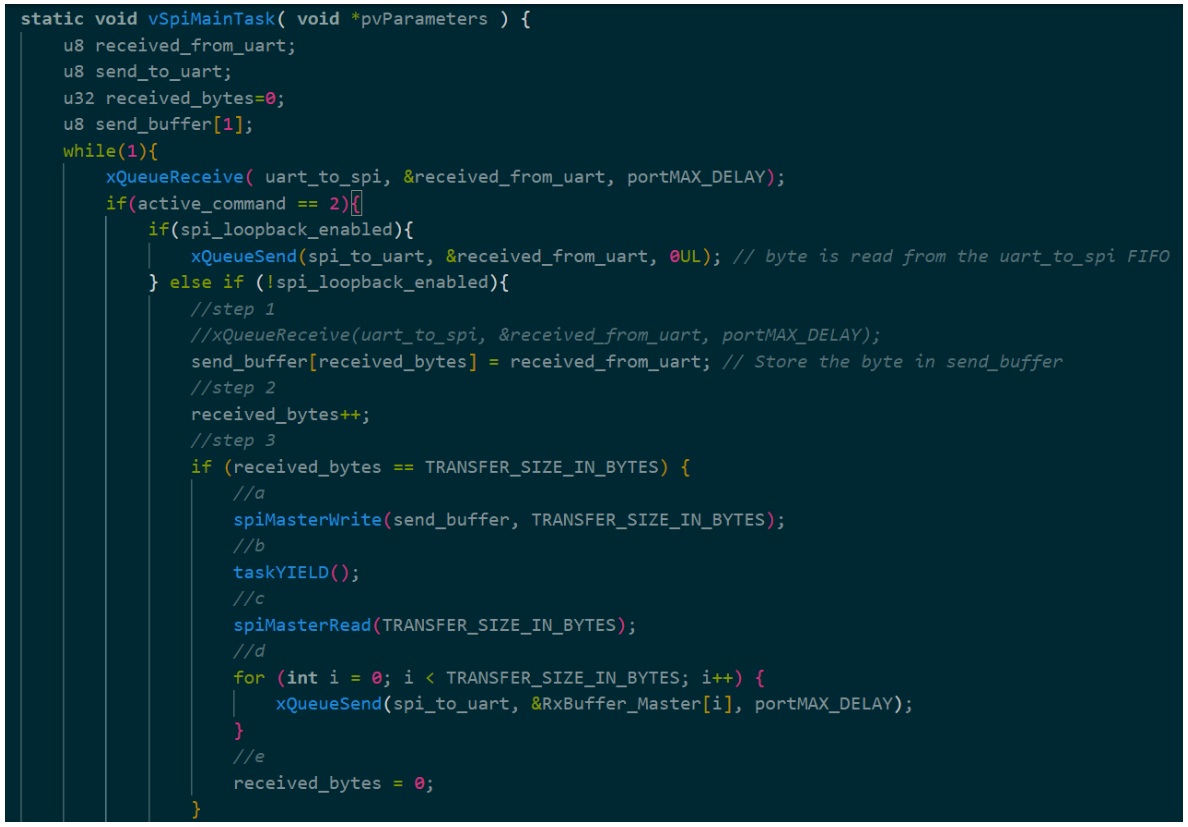


Figure 3.1.1 - Block diagram of a Zynq-7000 Processing System featuring SPI0 and SPI1: The 2 SPIs are used for high-speed serial communication with external peripherals. SPI0 and SPI1 provide dedicated SCLK, MOSI, MISO, and SS connections, enabling independent data exchange with multiple SPI devices. One SPI interface connects to the PmodOLED display via the AXI Lite SPI module, which bridges the AXI Interconnect and SPI protocol for efficient data transfer. These connections enhance system flexibility, allowing seamless integration of SPI-based peripherals while maintaining structured communication through the AXI bus.



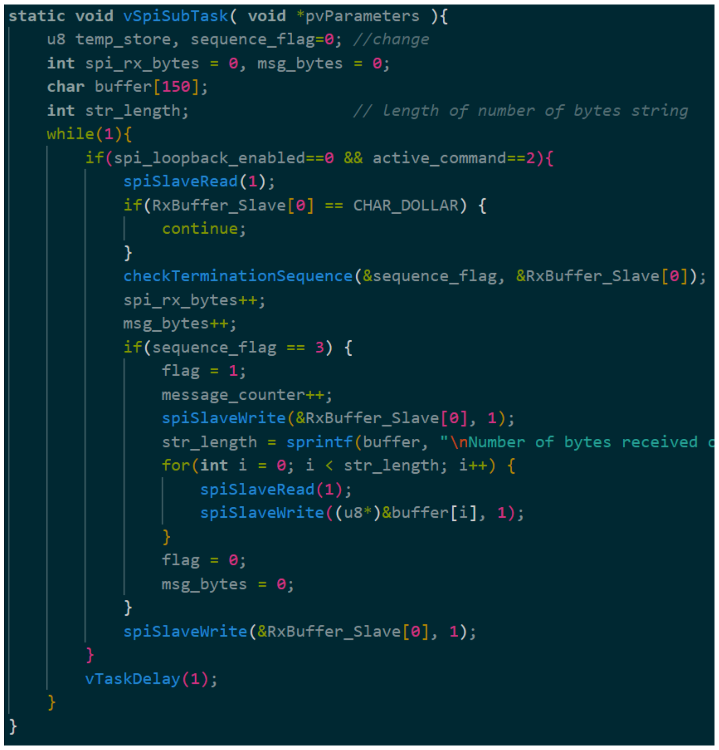
```
        //step 1
        xQueueSend(uart_to_spi, &dummy, 0UL);
        //step 2
        xQueueReceive(spi_to_uart, &spi_data, portMAX_DELAY);
        //step 3
        if(spi_data == dummy){
            break;
        }
        //step 5
        if(spi_data == '\0'){
            break;
        }
        //step 4
        while (XUartPs_IsTransmitFull(UART_BASE_ADDRESS) == TRUE) {
            taskYIELD();
        }
        XUartPs_WriteReg(UART_BASE_ADDRESS, XUARTPS_FIFO_OFFSET, spi_data);
    }
    terminateInput();
```

Figure 3.1.2 - SPI-UART Data Transfer Loop: loop that facilitates data transfer between SPI and UART in SPI1-SPI0 mode. When the flag is set, it initiates communication by sending a dummy character, waits for incoming SPI data, and transmits it to UART. The loop continues until a termination character (`'$'`) or a null character (`'\0'`) is received. The UART transmit buffer is checked to ensure it is not full before writing data, preventing buffer overflow.



```
static void vSpiMainTask( void *pvParameters ) {
    u8 received_from_uart;
    u8 send_to_uart;
    u32 received_bytes=0;
    u8 send_buffer[1];
    while(1){
        xQueueReceive( uart_to_spi, &received_from_uart, portMAX_DELAY);
        if(active_command == 2){
            if(spi_loopback_enabled){
                xQueueSend(spi_to_uart, &received_from_uart, 0UL); // byte is read from the uart_to_spi FIFO
            } else if (!spi_loopback_enabled){
                //step 1
                //xQueueReceive(uart_to_spi, &received_from_uart, portMAX_DELAY);
                send_buffer[received_bytes] = received_from_uart; // Store the byte in send_buffer
                //step 2
                received_bytes++;
                //step 3
                if (received_bytes == TRANSFER_SIZE_IN_BYTES) {
                    //a
                    spiMasterWrite(send_buffer, TRANSFER_SIZE_IN_BYTES);
                    //b
                    taskYIELD();
                    //c
                    spiMasterRead(TRANSFER_SIZE_IN_BYTES);
                    //d
                    for (int i = 0; i < TRANSFER_SIZE_IN_BYTES; i++) {
                        xQueueSend(spi_to_uart, &RxBuffer_Master[i], portMAX_DELAY);
                    }
                    //e
                    received_bytes = 0;
                }
```

Figure 3.1.3 - SPI Main Task for UART-SPI Communication: This code defines `vSpiMainTask`, which manages data exchange between UART and SPI. It receives data from the UART manager task via a queue and processes it based on the active command. If SPI loopback mode is enabled, the received byte is directly sent back to UART. Otherwise, the task buffers incoming bytes, transmits them over SPI once a complete message is received, and retrieves the response from SPI before forwarding it to UART.



```
static void vSpiSubTask( void *pvParameters ){
    u8 temp_store, sequence_flag=0; //change
    int spi_rx_bytes = 0, msg_bytes = 0;
    char buffer[150];
    int str_length;                      // length of number of bytes string
    while(1){
        if(spi_loopback_enabled==0 && active_command==2){
            spiSlaveRead(1);
            if(RxBuffer_Slave[0] == CHAR_DOLLAR) {
                continue;
            }
            checkTerminationSequence(&sequence_flag, &RxBuffer_Slave[0]);
            spi_rx_bytes++;
            msg_bytes++;
            if(sequence_flag == 3) {
                flag = 1;
                message_counter++;
                spiSlaveWrite(&RxBuffer_Slave[0], 1);
                str_length = sprintf(buffer, "\nNumber of bytes received c
                for(int i = 0; i < str_length; i++) {
                    spiSlaveRead(1);
                    spiSlaveWrite((u8*)&buffer[i], 1);
                }
                flag = 0;
                msg_bytes = 0;
            }
            spiSlaveWrite(&RxBuffer_Slave[0], 1);
        }
        vTaskDelay(1);
    }
}
```

Figure 3.1.4 - SPI Sub Task for SPI Message Handling and Monitoring: This code defines `vSpiSubTask`, which manages SPI communication by reading incoming bytes from the SPI slave node, tracking received data, and detecting a termination sequence (`\r%\r`). When the sequence is detected, it constructs a summary message containing the total bytes received, the last message size, and the total messages processed. The message is then sent back to the SPI master while ensuring all received bytes are echoed. The task also filters out dummy characters (`$`) and resets counters after each complete message.

Part 2

In Part 2 of this Lab, a modified version of the game "Space Invaders" was designed and implemented on the **Xilinx/AMD Pmod OLED**. The application used a mixture of polling and task queues to handle the execution of various functions on the device such as moving the player object to the designated row, shifting the enemies and attack beams across the screen, randomly spawning new enemies and periodically generating attacks, etc. The general control flow of the application is as such:

1. Draw the player object (displayed as a square) at the bottom centre of the screen. Use buttons 1 and 2 to traverse the screen left and right respectively
2. Periodically generate attack beamson the in the column the player object is currently occupying
3. Periodically generate enemy objects (displayed as squares) at the top of the screen with asymmetry between each object. Each enemy is spawned in a random column. Use a rolling pointer to keep track of the first empty positions within the array.
4. Update the screen periodically with the new or sustained position of each object with collision turned on. Use a rolling pointer to keep track of the empty positions within the array.
5. Traverse only the displayed enemy objects towards the bottom of the screen. When an object's x position is within a defined threshold of the bottom of the screen. Execute the game over routine

by updating the state variable. All tasks aside from the updateScreen tasks Yield when when the gameOver variable is true.

6. Traverse only the displayed attacks towards the top of the screen. When any attack's x position is within a defined threshold of the x position for an enemy and within the same column, turn the collision off for both objects, signifying destruction after colliding. Furthermore, update the current score and notify the user by sending the message to toggle on the RGB LED if the conditions for a power up are met.When an attack exits the range of the screen, turn it's collision off automatically. Additionally, with each enemy that is destroyed, increase the movement refresh speed of the game up to a maximum value.
7. When button 4 is pressed by the user, send the message through the respective queue to execute the power up task, essentially, "Clear the screen of all enemies" allowing the user to avoid the game over state. This action will only succeed if the RGB state was active prior to execution. Toggle off the RGB LED to signify that the power up has been used.
8. When the game over state has been entered, print a message on the OLED that includes the user's score. Simultaneously, clear all objects from the screen, asides from the player, by setting their collision off and reset the position for each object to starting points. Turn off the RGB LED if it was previously on and reset the game speed back to the starting speed.

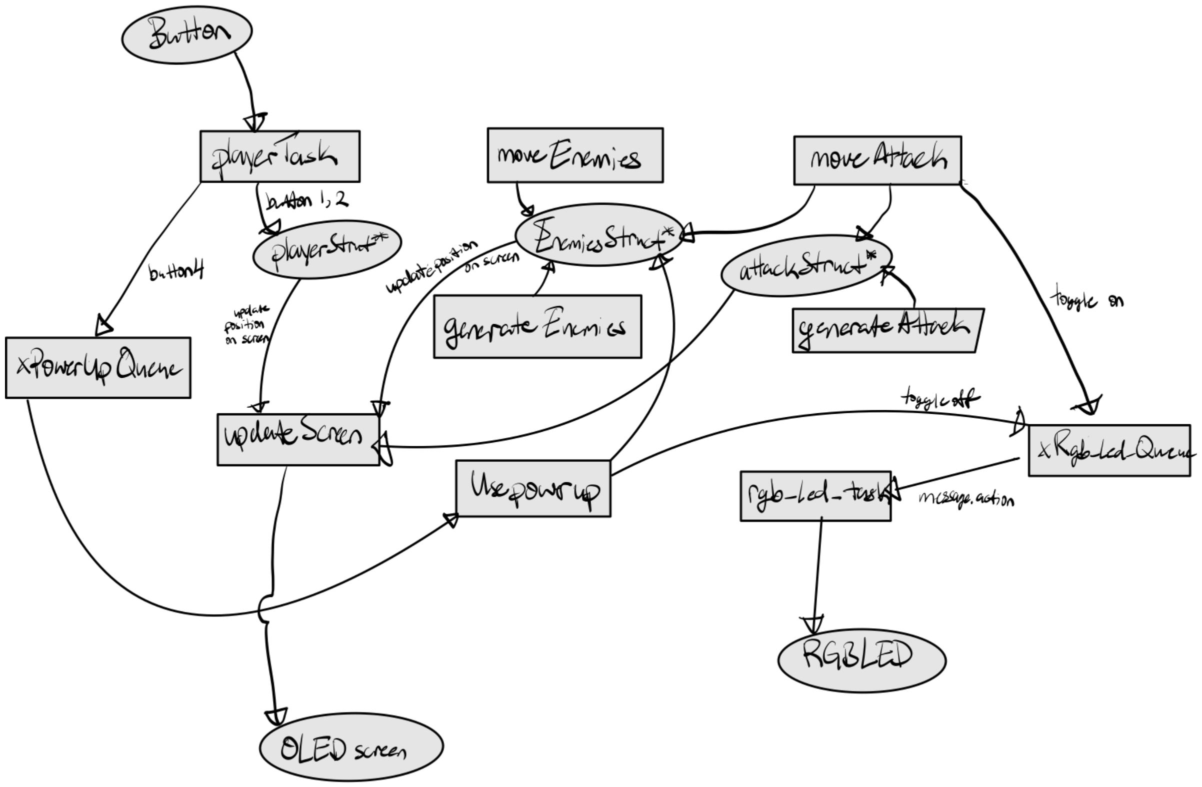
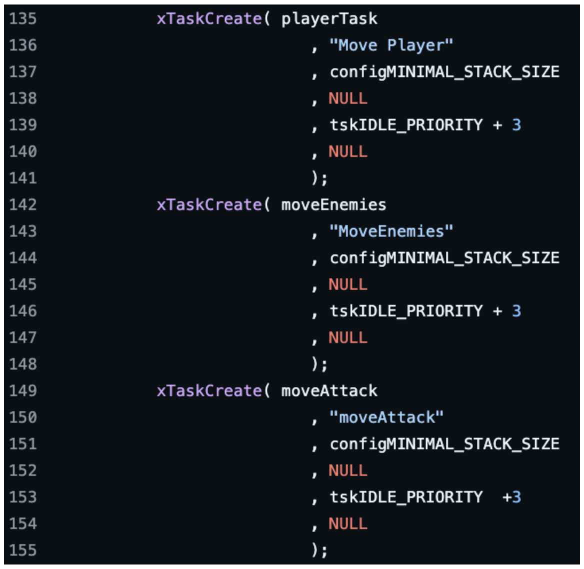
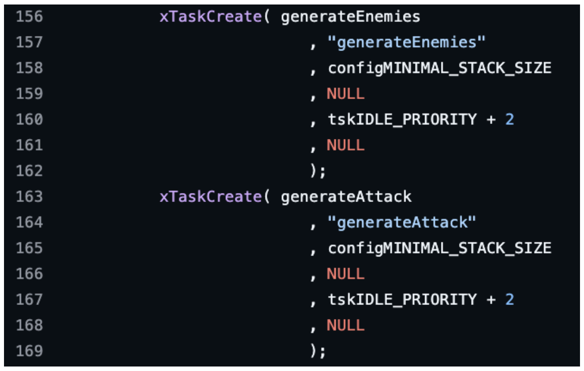


Figure 3.2.1 - Task Flow Diagram: Architecture of the "Space Invader" game. Each task (rectangles) communicates with the updateScreenTask through the use of globally shared objects of the user defined `GameObject` type. Periodically, the updateScreen Task refreshes the display with the new positions for each object. Additionally, tasks communicate with each other on demand through the `xRgbLedQueue` and `xPowerUpQueue` to notify the user when their power up is available and use their power up respectively.



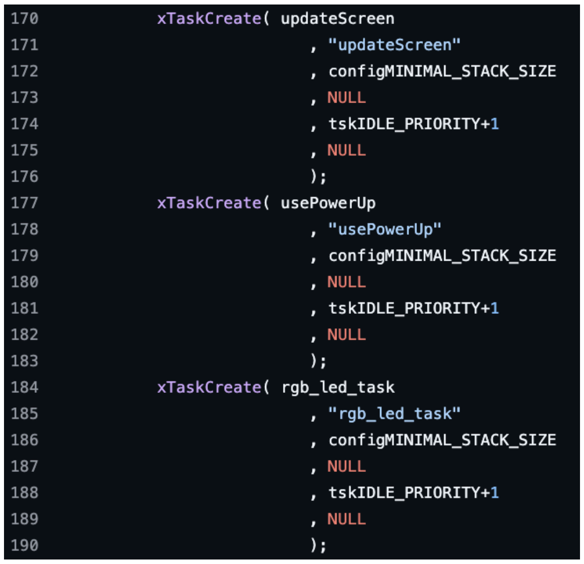
```
        xTaskCreate( playerTask
                        , "Move Player"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY + 3
                        , NULL
                        );
        xTaskCreate( moveEnemies
                        , "MoveEnemies"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY + 3
                        , NULL
                        );
        xTaskCreate( moveAttack
                        , "moveAttack"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY  +3
                        , NULL
                        );
```

3.2.2a



```
        xTaskCreate( generateEnemies
                        , "generateEnemies"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY + 2
                        , NULL
                        );
        xTaskCreate( generateAttack
                        , "generateAttack"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY + 2
                        , NULL
                        );
```

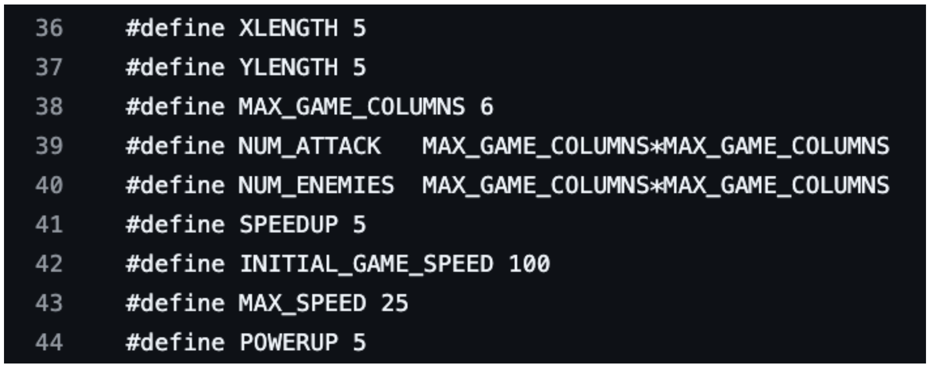
3.2.2c



```
        xTaskCreate( updateScreen
                        , "updateScreen"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY+1
                        , NULL
                        );
        xTaskCreate( usePowerUp
                        , "usePowerUp"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY+1
                        , NULL
                        );
        xTaskCreate( rgb_led_task
                        , "rgb_led_task"
                        , configMINIMAL_STACK_SIZE
                        , NULL
                        , tskIDLE_PRIORITY+1
                        , NULL
                        );
```

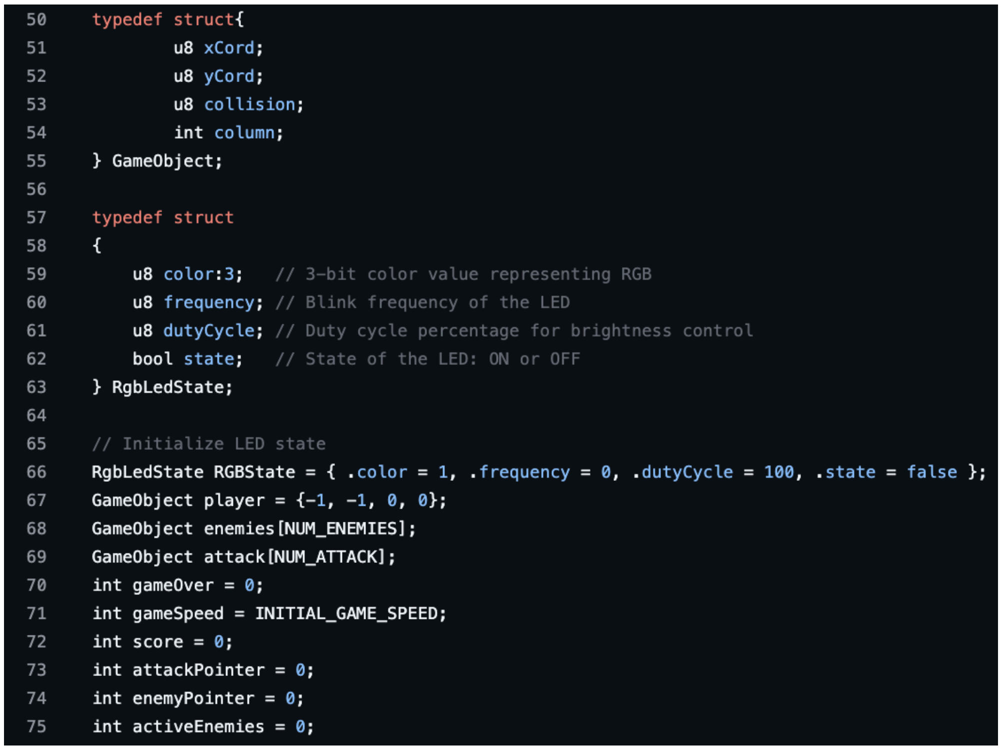
3.2.2c

Figures 3.2.2 - Task Priority Definitions: Code snippets detailing the priority levels for the different tasks. Tasks with the responsibility of managing the movement of objects on the screen were given the highest priority as to ensure smooth and consistent response times. Tasks with the responsibility of generating new objects were given mid level priority as to ensure adequate CPU time. Low priority tasks include the roles of updating the screen, using the power up, and lighting up the RGB LED as these tasks either happen infrequently or are not as critical as the aforementioned tasks.



```
#define XLENGTH 5
#define YLENGTH 5
#define MAX_GAME_COLUMNS 6
#define NUM_ATTACK   MAX_GAME_COLUMNS*MAX_GAME_COLUMNS
#define NUM_ENEMIES  MAX_GAME_COLUMNS*MAX_GAME_COLUMNS
#define SPEEDUP 5
#define INITIAL_GAME_SPEED 100
#define MAX_SPEED 25
#define POWERUP 5
```

Figure 3.2.3 - Configuration of game constants: Constants defined to control the games settings and difficulty. `XLENGTH` and `YLENGTH` are the size of the player and enemy objects in columns and rows respectively. This value was chosen such that each square object would have `MAX_GAME_COLUMNS` different positions on the y axis. Furthermore , the maximum number of attack and enemy objects that can be tracked and displayed on the screen at any one time was set arbitrarily here. The rate at which the movement of objects on the screen will begin to speed up as well as the number of enemies to destroy before the power up is given can be configured here for different difficulty levels.

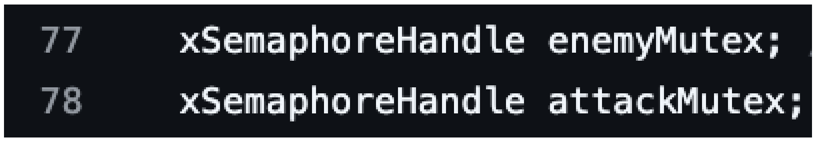


```c
typedef struct{
        u8 xCord;
        u8 yCord;
        u8 collision;
        int column;
} GameObject;

typedef struct
{
    u8 color:3;   // 3-bit color value representing RGB
    u8 frequency; // Blink frequency of the LED
    u8 dutyCycle; // Duty cycle percentage for brightness control
    bool state;   // State of the LED: ON or OFF
} RgbLedState;

// Initialize LED state
RgbLedState RGBState = { .color = 1, .frequency = 0, .dutyCycle = 100, .state = false };
GameObject player = {-1, -1, 0, 0};
GameObject enemies[NUM_ENEMIES];
GameObject attack[NUM_ATTACK];
int gameOver = 0;
int gameSpeed = INITIAL_GAME_SPEED;
int score = 0;
int attackPointer = 0;
int enemyPointer = 0;
int activeEnemies = 0;
```

Figure 3.2.4 - Game state variables and object declarations: Details of the different state variables used within the application. Objects displayed on the screen are of type `GameObject` to enforce consistency with the way they are handled in later functions. The `.collision` field dictates which objects are to be displayed on the screen. Moreover, integer type variables are used to keep track of the internal state of the game. That is, how fast to update movement, the location of available objects within each array, and the play state of the game.



```c
xSemaphoreHandle enemyMutex;
xSemaphoreHandle attackMutex;
```

Figure 3.2.5 - Synchronization: Declaration of binary mutexes to synchronize tasks that manipulate the enemy and attack arrays. These mutexes ensure priority inversion, such that the “GenerateX” tasks at priority 2 can complete without interruption when given the chance. These mutexes are primarily used to handle the reading and writing of the “collision” field as this is the field that moderates access to locations within the GameObject arrays. The exception, however, is the usePowerUp task. Because it was assumed for this simple application that this short low priority task would primarily be running in between the

movement delays. A more robust synchronization scheme would need to be adopted along with further complexities of the application.

Questions:

1. **How does activating the loopback mode in both (UART and SPI) impact the interaction and data flow between vUartManagerTask, vSpiMainTask and vSpiSubTask?**
   Activating loopback mode in both UART and SPI fundamentally alters the flow of data and the interactions between `vUartManagerTask`, `vSpiMainTask`, and `vSpiSubTask`. When UART loopback is enabled (`uart_loopback_enabled=1`), `vUartManagerTask` directly echoes received characters back to the terminal without passing them to SPI. When SPI loopback is enabled (`spi_loopback_enabled=1`), the SPI0-SPI1 connection is disabled, and instead of data transferring between the master (`vSpiMainTask`) and slave (`vSpiSubTask`), the received SPI data is simply sent back through the `spi_to_uart` queue for direct UART output. In this case, `vSpiSubTask` does not contribute to the data processing flow since no SPI transactions occur. If both loopbacks are enabled simultaneously, UART input gets echoed back immediately by `vUartManagerTask`, and the SPI subsystem does not process any meaningful data, effectively isolating UART from SPI. This means that `vSpiMainTask` and `vSpiSubTask` do not exchange real messages, and `spi_to_uart` merely forwards UART input instead of an SPI response.

2. **What difficulties did you encounter while setting up SPI communication between the main and sub devices, and what solutions did you implement?**
   a. Initially, we observed garbled or warbled text being output in the SDK Terminal. After some troubleshooting, we identified that the issue was likely caused by either a buffer overflow or improper handling of the FIFO queue.
      i. Solution: We tackled this problem by improving the handling of the FIFO queue. Specifically, we leveraged the vUartManagerTask to ensure that data was being read and written to the queue correctly. By synchronizing the task responsible for managing UART communication with the FIFO operations, we were able to ensure that data flow was more controlled, preventing any overflow and ensuring clear output.
   b. Another significant issue we faced was that the subtask was not reading any input. This prevented us from even properly terminating the task, as it couldn't recognize numbers or interpret the termination sequence. Initially, we suspected the problem could be related to the task's inability to process input from the SPI communication or issues with the way input was being parsed.
      i. Solution: After thorough inspection, we realized that part of the subtask's code was incomplete, and there were issues with how the input was being handled. We hadn't properly implemented certain steps in the subtask's code, which led to its failure to read the input correctly. To resolve this, we finished the missing steps in the subtask's code, ensuring that the input was properly received and processed. Additionally, we improved the queue handling to ensure that the main task and

subtask communicated effectively and that data was properly queued for processing. This adjustment allowed the subtask to successfully read inputs, including terminating signals, and perform its intended function.

3. **Describe the challenges that you faced when interfacing with the Pmod OLED screen and describe how you resolved them.**
Primarily, we faced challenges updating the position of the player object. We noticed two problems. The first is that randomly the player object would jump to the left most column, and secondly that users trying to traverse outside of the bounds sporadically caused the object to teleport and get stuck on the right side of the screen. The first issue was resolved by refactoring the code to use a regular object instead of a pointer to a game object, and then modifying the function calls to pass the address of the object rather than the value of the pointer. We suspected some type of memory overlap was causing the yCord attribute to be modified by an unknown line of code. This change resolved this issue. Secondly, rather than using the same variable for displaying and modifying the column of an object (yCord), we introduced a new field `column` that would be used within mutex semaphores to modify the column. This standardized the position of the objects because we no longer had to calculate the offset of the object in rows each time a button was pressed, rather we could increment/decrement by 1. Then, when the updateScreen task was called, we would calculate the position in rows at runtime and display it. This allowed us to not only debug and read the state of objects easier, but also simplified the code allowing us to track down the error that caused out of bound movement: inclusive rather than exclusive checking within an if statement which allowed yCord to be updated outside of the bounds leading to unexpected behavior that could not be recovered with the current structure.

4. **Does input from the peripherals affect the dynamic graphics on the Pmod OLED screen? Give examples from your game.**
Yes, input from the peripherals does affect the dynamic graphics on the Pmod OLED screen. For example, the buttons controlling the movement of the "character" to go left and right directly alter its position on the screen. Additionally, the button that activates a super power-up and kills all the enemies results in a change in the game's graphics, as it removes the enemies from the display. This shows how user input directly influences the visual updates on the Pmod OLED.

5. **Reflecting on this lab's activities, how have the tasks and experiments deepened your understanding of designing embedded systems and operating within a real-time OS framework?**
This lab deepened our understanding of embedded system design and working within a real-time OS framework by providing hands-on experience with SPI communication, task synchronization, and resource management. We gained insights into how tasks are scheduled and prioritized in an RTOS, and the importance of managing data flow through queues and buffers to avoid issues like overflow and corrupted communication. Debugging real-time constraints, like input reading failures and timing issues, emphasized the critical role of precise task coordination and efficient hardware-software integration. Overall, the lab reinforced the complexity of designing reliable embedded systems that meet real-time requirements and highlighted the need for careful synchronization and resource management.

Testing Suite:

Tests	***Expected Result***	***Actual Result***	***Rationale***
Part 1			
Test UART transmission	When you enter 1 to the terminal, the UART should be enabled, and any entered text should be echoed to the terminal. When you enter 1 again, the transmission is disabled.	**When you enter 1, the RGB LED glows red and any text that is entered is echoed to the terminal. Sending 1 again disables the UART and the LED turns off. This also happens when the termination sequence (/r%/r) is entered**	Ensuring that the UART transmission was working as expected.
Testing SPI Main task transmission	When you enter 2 the SPI should be enabled, any text that is sent should be echoed back to the terminal, and entering the termination sequence disables the SPI and returns info about bytes passed.	**When you enter 2, the RGB LED glows blue and any text that is entered is echoed to the terminal. If the termination sequence is passed, the following info is returned: Number of bytes received over SPI, Last message byte count, and Total messages received.**	Ensuring that the SPI Main Task was working as expected.
Testing SPI subtask transmission	After the SPI Main task is enabled, hitting 2 again will disable the main task and switch to the subtask. any text that is sent should be echoed back to the terminal, and entering the termination sequence disables the SPI and returns info about bytes passed.	**When you enter 2, the RGB LED glows green, the main task is disabled and the sub task is enabled. Any text that is entered is echoed to the terminal. If the termination sequence is passed, the following info is returned: Number**	Ensuring that the SPI Sub Task was working as expected and communicating with the Main Task properly.

		of bytes received over SPI, Last message byte count, and Total messages received. Hitting 2 again totally disables the SPI	
Testing message and byte counter	After running either SPI task, entering the termination sequence will return the following info: Number of bytes received over SPI, Last message byte count, and Total messages received.	**When the termination sequence is entered after running the SPI, it will return the following info:Number of bytes received over SPI, Last message byte count, and Total messages received. Number of bytes is equivalent to characters plus enters.**	To ensure that both SPI were communicating properly, that the count was consistent, and that the values were being counted properly.
Part 2			
Tests player movement	When button 1 or 2 is pressed, the player should be displayed 1 column position to the left or right	**Player shifts to the left when Button 1 is pressed; Player shifts to the right when Button 2 is pressed**	Ensuring the action occurs with a reasonable response time correctly.
Test boundary movement	When button 1 is pressed in the left most column or button 2 is pressed in the right most column, do nothing	**Players position does not change**	Ensures the logical correctness of movement.
Test Power Up execution	When button 3 is pressed, all enemies disappear from the screen immediately. RGB LED turns off	**All enemies disappear from the screen instantly and RGB LED is turned off. Access to shared array does not cause display problems**	Ensure the execution of the low priority task still has a reasonable response time
Test collision for	When an attack hits an enemy, both objects	**Attacks successfully**	Ensures enemy destruction is successfully modeled within

attack beam and enemies	disappear from the screen and will not affect game play	**destroy enemies. Gameplay continues as the enemy is not treated as reaching the bottom**	the system by setting collision to 0.
Test continuous fire by disabling enemies	Attacks are periodically and predictably streamed up the screen	**Attacks continue but begin to stop after a few seconds**	Confirms the need to properly recycle attack and enemy array positions by setting their collision to 0 when they go off screen
Test Game over when enemies get to low by disabling attack	When an enemy gets to the bottom of the screen, execute the "restart game" protocol	**Enemies reach the bottom of the screen, "Game over Total Score: 0" is displayed and the game starts again**	Ensures the application states are maintained correctly and can be reset to initial conditions without error
Test RGB power up	Notify the user that a power up is available every 5 enemies destroyed	**RGB LED turns on CYAN after 5 enemies have been destroyed**	Test communication between tasks and the successful execution of the RGB task
Test consecutive play	When a game finishes another game will start initialized to the same conditions as the prior	**Regardless of score or position several games are able to execute without error from the same initial state**	Ensures proper management of state variables, data, and task execution between iterations of the game

Conclusion

The purpose of this lab was to explore the Serial Peripheral Interface (SPI) using the Zybo Z7 and gain practical experience with SPI communication in both master and slave configurations. Initially, a loopback mode was implemented to establish communication between SPI0 and SPI1, allowing data to be transmitted and received within the system. The SPI tasks were integrated into a FreeRTOS environment, ensuring efficient data exchange between vUartManagerTask, vSpiMainTask, and vSpiSubTask.

Next the SPI interface was extended to interact with the Pmod OLED display, demonstrating the process of sending graphical data over SPI. This portion of the lab provided valuable practice in embedded graphics programming and interfacing with SPI peripherals. By integrating input controls, we were able to dynamically update the display in real-time, further reinforcing our understanding of real-time system design and task synchronization.

Lastly, challenges such as SPI data inconsistencies and display synchronization were encountered and addressed through improved queue management and refined task execution strategies. Through observation and testing, it was concluded that properly structured SPI communication and FreeRTOS task management are crucial for real-time embedded applications. This lab provided insight into the complexities of embedded system development, highlighting the importance of efficient data handling, peripheral integration, and real-time processing in modern computing applications.

Appendix

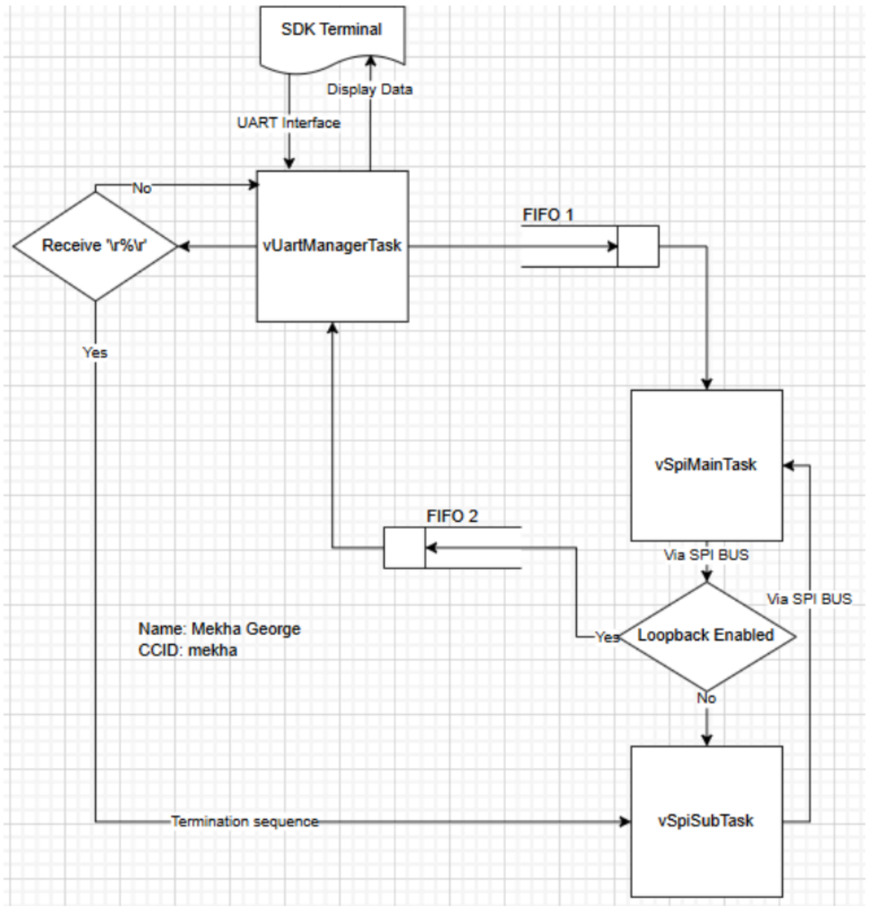


Figure 3.3.1 - Mekha George Prelab Data Flow diagram

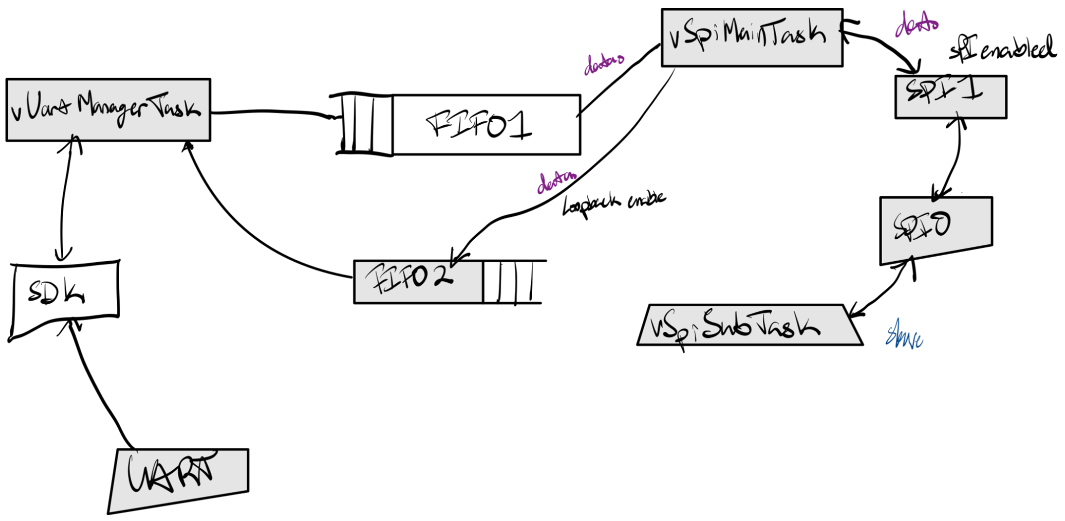


Figure 3.3.2 - Timmy Ifidon Prelab Data Flow diagram