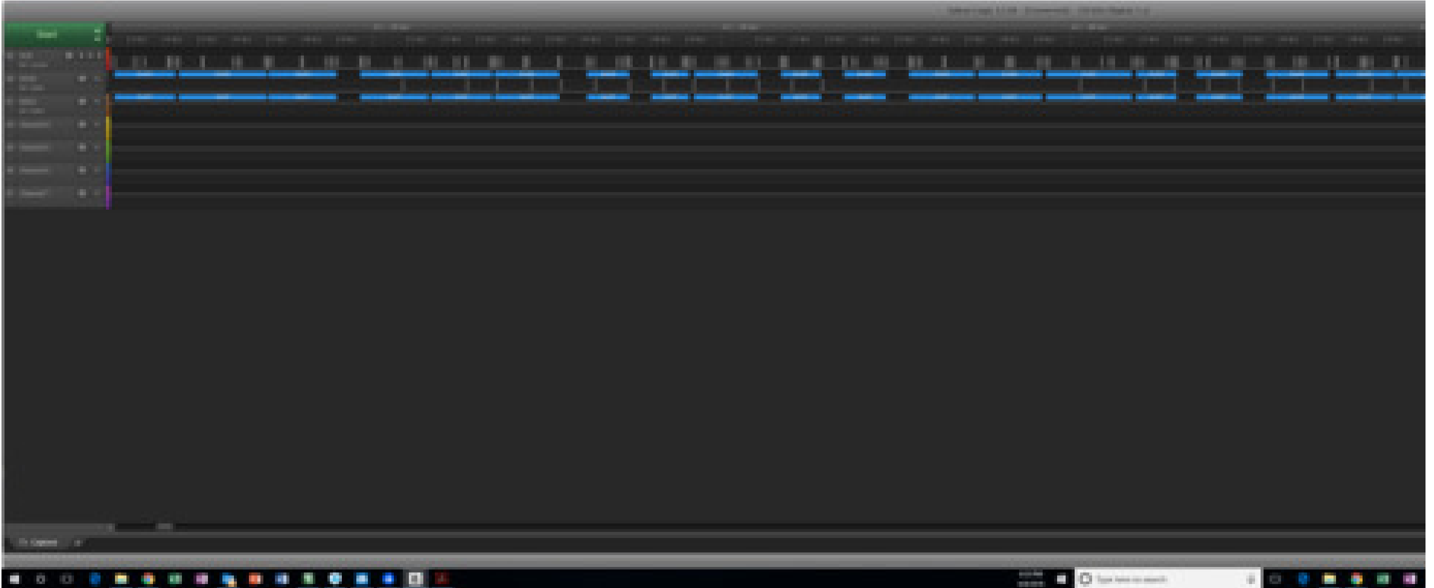
Hello, I'm working on a project involving STM32L476RG Microcontroller. I'm trying to write a piece of code for creating an SPI Driver for my project and seem to have some issues. I'm using Keil v5 IDE. The output doesn't seem to be close to my expected behavior of the Microcontroller.

My logic analyzer output is below. My clock is irregular and MOSI has garbage data. Anybody with any leads, please do let me know. Thanks in advance for your valuable time.



My piece of code:

```
#include <stdint.h>
#include "stm32l4xx.h"                  // Device header
#include <stdbool.h>

volatile uint8_t* txBufferPtr;          // tx buffer pointer
uint16_t txBufferLength;                    // tx buffer size
volatile uint16_t txBufferCounter;     // tx buffer counter
volatile uint8_t* rxBufferPtr;          // rx buffer pointer
uint16_t rxBufferLength;                    // rx buffer size
volatile uint16_t rxBufferCounter;     // rx buffer counter
uint8_t datasize;          // selected data transfer size [bits]
bool master;               // whether SPI is master or slave


/**
 * @brief     Transfer data over SPI.
 * @param     channel - 0-based index of the SPI port of interest
 * @param     *txArray - pointer to the transmit buffer
 * @param     *rxArray - pointer to the receive buffer
 * @param     length - number of bytes to transmit
 */
void ST_SPI_Transfer( uint8_t *txArray, uint8_t *rxArray, uint32_t length)
{
     // Set up transmit buffer.
     txBufferPtr = txArray;
     txBufferLength = length;
     txBufferCounter = length;

     // Set up global receive variables.
     rxBufferPtr = rxArray;
     rxBufferCounter = length;
     rxBufferLength = length;

     // Read data register once before enabling the receive interrupt.
     // This makes sure the data register is empty and avoids spurious interrupts.
     uint32_t val = SPI2->DR;

     // This prevents a compiler warning about var being an unused variable.
     (void)val;

     // Enable receive interrupt.
     SET_BIT(SPI2->CR2, SPI_CR2_RXNEIE);

     // Enable transmit interrupt.
     // This will generate an interrupt as soon as the SPI is enabled.
     SET_BIT(SPI2->CR2, SPI_CR2_TXEIE);

     // Enable the SPI module if its not enabled already.
     // This is done last to prevent spurious SPI activity.
     // If the peripheral is disabled...
     // (Checking prevents us from interrupting anything already going on.)
     if (!(SPI2->CR1 & SPI_CR1_SPE))
     {
          // Set the enable bit in Control Register 1.
          SET_BIT(SPI2->CR1, SPI_CR1_SPE);
     }
}

/**
 * @brief     Initialize a given SPI device.
 * @remarks     This must be called before anything else.
 * @param     channel - 0-based index of the SPI port of interest
```

```
 * @param     configPtr - structure pointer with settings
 */
void ST_SPI_Init()
{
    // Enable the clock for the selected SPI port.
     RCC->APB1ENR1 |= RCC_APB1ENR1_SPI2EN;

     // Configure phase and polarity (a.k.a. SPI mode).
     CLEAR_BIT(SPI2->CR1, SPI_CR1_CPOL);
     SET_BIT(SPI2->CR1, SPI_CR1_CPHA);

     // Set the master mode bit in Control Register 1.
     SET_BIT(SPI2->CR1, SPI_CR1_MSTR);

     // Configure format for 1 byte transfer.
     MODIFY_REG(SPI2->CR2, SPI_CR2_DS, 0x07);

     // Enable Software Slave Management bit.
     SET_BIT(SPI2->CR1, SPI_CR1_SSM);

     // Set the Slave Select pin high.
     SET_BIT(SPI2->CR1, SPI_CR1_SSI);

     // Configure speed.
     MODIFY_REG(SPI2->CR1, SPI_CR1_BR, 4);

     // Set direction.
     //SET_BIT(SPI2->CR1, SPI_CR1_BIDIOE);

     //Set LSBFirst
     SET_BIT(SPI2->CR1, SPI_CR1_LSBFIRST);

     //Set NSSP Bit
     SET_BIT(SPI2->CR2, SPI_CR2_NSSP);

     // Enable the appropriate IRQ handler in the NVIC.
     NVIC_EnableIRQ(SPI2_IRQn);
}

int main(void)
{

     uint8_t readData[4];                              // data received from slave
     uint8_t writeData[] = { 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9}; // data to send to slave

     // Configure SPI clock using GPIOB PIN 13.
     RCC->AHB2ENR = (1 << 1);             //Initialize Port
    GPIOB->MODER &= ~(3 << (2 * 13));
     GPIOB->MODER |= (2 << (2 * 13));                     //Alternate Function mode
     GPIOB->OSPEEDR &= ~(3 <<(2 * 13));
     GPIOB->OSPEEDR |= (2 << (2 * 13));                  //Medium Speed
     GPIOB->OTYPER &= ~(1 << 13);
     GPIOB->OTYPER |= (2 << 13);                    //Output Type
     GPIOB->AFR[1] |= (5 << (5*4));
     GPIOB->PUPDR = 2;                       //Pull Down


     // Configure SPI MISO pin using GPIOB PIN 14.
     // Set settings common to all GPIO pins used in the test.
     GPIOB->MODER &= ~(3 << (2 * 14));
     GPIOB->MODER |= (2 << (2 * 14));                    //Alternate Function mode
     GPIOB->OSPEEDR &= ~(3 <<(2 * 14));
     GPIOB->OSPEEDR |= (2 << (2 * 14));                  //Medium Speed
     GPIOB->OTYPER &= ~(1 << 14);
     GPIOB->OTYPER |= (2 << 14);                    //Output Type
     GPIOB->AFR[1] |= (5 << (6*4));
     GPIOB->PUPDR = 1;

     // Configure SPI MOSI pin using GPIOB PIN 15.
     GPIOB->MODER &= ~(3 << (2 * 15));
     GPIOB->MODER |= (2 << (2 * 15));                    //Alternate Function mode
     GPIOB->OSPEEDR &= ~(3 <<(2 * 15));
     GPIOB->OSPEEDR |= (2 << (2 * 15));                   //Medium Speed
     GPIOB->OTYPER &= ~(1 << 15);
     GPIOB->OTYPER |= (2 << 15);
    GPIOB->AFR[1] |= (5 << (7*4));
     GPIOB->PUPDR = 1;

     // Initialize the SPI peripheral.
     ST_SPI_Init();

     // If the SPI bus is busy...
     while (READ_BIT(SPI2->SR, SPI_SR_BSY));

     while(1)
     {
         //Transfer Data to slave
          ST_SPI_Transfer( writeData, readData, sizeof(readData));

          //Wait for the data to send
          while(READ_BIT(SPI2->SR, SPI_SR_BSY));
     }

}

void SPI2_IRQHandler(void)
{
     // If the RXNE is enabled and its flag is set...
     if ((SPI2->SR & SPI_SR_RXNE) && (SPI2->CR2 & SPI_CR2_RXNEIE))
     {
          // Receive data.
          // If we're using 8-bit data transfer size...
          if (datasize <= 8)
          {
```

```
                if (rxBufferPtr)
                {
                     *(rxBufferPtr) = SPI2->DR;
                }

                // Increment the pointer by one.
                rxBufferPtr++;

                // Decrement the byte counter by one.
                rxBufferCounter--;
           }

           // If we have no more data to transfer...
           if (rxBufferCounter == 0)
           {
                // Wait for the SPI peripheral to finish any transmission.
                while (READ_BIT(SPI2->SR, SPI_SR_BSY))

                // Disable receive interrupt.
                CLEAR_BIT(SPI2->CR2, SPI_CR2_RXNEIE);
           }
      }
      // If the TXE is enabled and its flag is set...
      else if ((SPI2->SR & SPI_SR_TXE) && (SPI2->CR2 & SPI_CR2_TXEIE))
      {
           // Transmit data.
           // Transmit one byte.
           if (datasize <= 8)
           {
                SPI2->DR = *(txBufferPtr);

                // Increment the data pointer.
                txBufferPtr++;

                // Decrement the counter of bytes left to read.
                txBufferCounter--;
          }
           // If we have no more data to transfer...
           if (txBufferCounter == 0)
           {
                // Disable the transmit interrupt.
                CLEAR_BIT(SPI2->CR2, SPI_CR2_TXEIE);
           }
     }
}
```

??