Board-to-Board Communication using SPI between STM32 and Rugged Board

This document provides detailed information on establishing board-to-board communication between an STM32 microcontroller and a Rugged Board using the SPI (Serial Peripheral Interface) protocol. The goal is to configure the STM32 as the SPI slave and the Rugged Board as the SPI master to facilitate communication between the two boards.

Introduction:-

In embedded systems, effective communication between boards or microcontrollers is essential for the exchange of data, control signals, or even commands. One such method of communication is the **Serial Peripheral Interface (SPI)**, a synchronous serial communication protocol designed for short-distance communication. In this document, we explore the implementation of SPI communication between an **STM32** board (acting as the slave) and a **Rugged Board** (acting as the master). The STM32 communicates using its **SPI1** pins, while the Rugged Board operates as the master using the **MikroBus SPI pins**.

System Overview:-

The **STM32** and **Rugged Board** will communicate using the **SPI** protocol. The STM32 will be configured as the **SPI slave**, and the Rugged Board will act as the **SPI master**. The Rugged Board will control the clock (SCK) and initiate communication, sending data to the STM32. In turn, the STM32 will receive data sent from the Rugged Board and may also send responses back if necessary.

Key Components:-

- STM32 microcontroller (Slave)
- Rugged Board (Master)
- SPI Pins: MOSI, MISO, SCK, CS
- Data communication: Full-duplex data transmission between the boards.

Communication Protocol: SPI

SPI is a synchronous, full-duplex communication protocol that allows data to be transmitted and received simultaneously. The protocol relies on the following primary signals:

- **MOSI (Master Out Slave In)**: Data line for sending data from the master to the slave.
- **MISO (Master In Slave Out)**: Data line for receiving data from the slave to the master.
- **SCK (Serial Clock)**: Clock signal generated by the master to synchronize data transfer.
- **CS (Chip Select)**: Signal used to select the slave device for communication.

SPI Modes: SPI operates with various clock configurations:

- **CPOL (Clock Polarity)**: Defines whether the clock is idle high or low.
- **CPHA (Clock Phase)**: Defines whether data is captured on the rising or falling edge of the clock.

For this setup, the communication will be done using **SPI Mode 0**, where:

- **CPOL = 0** (Idle state low)
- **CPHA = 0** (Data captured on the rising edge)

This mode ensures synchronization between the Rugged Board (master) and the STM32 (slave) for smooth communication.

Hardware Requirements:-

The hardware setup involves connecting the SPI pins between the **STM32** and **Rugged Board**. Below are the pin connections for both boards:

STM32 SPI Pin Connections:

- CS (Chip Select): Pin D10
- MOSI (Master Out Slave In): Pin D11
- MISO (Master In Slave Out): Pin D12
- SCK (Serial Clock): Pin D13

Rugged Board SPI Pin Connections (MikroBus SPI Pins):

- MOSI (Master Out Slave In): Pin 3
- MISO (Master In Slave Out): Pin 4
- SCK (Serial Clock): Pin 5
- CS (Chip select): Pin6

Common Ground:

- Both boards share a common **ground (GND)**

Software Requirements:-

The software implementation will involve configuring the **STM32** as an SPI slave and the **Rugged Board** as an SPI master. The following key software components are required:

STM32 Software:

SPI Configuration as Slave:

- Set up the **SPI1** peripheral to function as an SPI slave.
- Configure the **MOSI**, **MISO**, **SCK**, and **CS** pins as per the STM32 datasheet and pinout diagram.
- Enable interrupts for receiving and sending data.

Data Reception:

- Set up interrupt-based or polling-based data reception on the **MISO** pin.
- Process the received data and send back any response if needed.

SPI Communication:

- Set up the necessary SPI initialization code to configure the communication settings.
- Enable the SPI interface to allow for data transmission.

GPIO Configuration:

- Configure the **Chip Select (CS)** pin as input to detect when the Rugged Board has selected the STM32 for communication.

Rugged Board Software (Master):

SPI Configuration as Master:

- Set up the SPI interface to act as a master.
- Configure the clock polarity, phase, and speed according to the STM32's slave configuration (SPI Mode 0).
- If `/dev/spidev0.0` is not present, it means the SPI interface is not enabled or the device tree is not configured correctly.
- pip install spidev

Data Transmission:

- Send data to the STM32 via **MOSI** and monitor the **MISO** line for any response from the STM32.
- Implement a polling mechanism or interrupt-based approach to handle communication.

SPI Initialization:

- Initialize the SPI peripheral with the proper clock settings, data frame format, and chip select control.

Chip Select (CS):

- Control the **CS** pin to enable communication with the STM32. When the **CS** pin is pulled low, it indicates that the STM32 is selected and communication can begin.

STM32 (Slave) Source Code:-

```c
/* USER CODE BEGIN Header */
/**

*****************************************************************************
*
 * @file           : main.c
 * @brief          : Main program body

*****************************************************************************
*
 * @attention
 *
 * Copyright (c) 2024 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE
file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *

*****************************************************************************
*
 */
/* USER CODE END Header */
/* Includes
------------------------------------------------------------------*/
#include "main.h"
#include <stdio.h>
```

```c
#include <string.h>
/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
/* USER CODE END Includes */
/* Private typedef
-----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */
/* Private define
------------------------------------------------------------*/
/* USER CODE BEGIN PD */
uint8_t buffer[100];  // Buffer for storing messages
uint8_t rxData[3];    // Buffer to store received data from SPI
uint8_t txData[] = {0x01, 0x02, 0x03}; // Data to send via SPI
/* USER CODE END PD */
/* Private macro
-------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */
/* Private variables
---------------------------------------------------------*/
QSPI_HandleTypeDef hqspi;
SPI_HandleTypeDef hspi1;
UART_HandleTypeDef huart2;
/* USER CODE BEGIN PV */
/* USER CODE END PV */
/* Private function prototypes
-----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_QUADSPI_Init(void);
static void MX_SPI1_Init(void);
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */
/* Private user code
---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* USER CODE END 0 */
/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
 /* USER CODE BEGIN 1 */
 /* USER CODE END 1 */
 /* MCU
Configuration--------------------------------------------------------*/
 /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
 HAL_Init();
```

```c
 /* USER CODE BEGIN Init */
 /* USER CODE END Init */
 /* Configure the system clock */
 SystemClock_Config();
 /* USER CODE BEGIN SysInit */
 /* USER CODE END SysInit */
 /* Initialize all configured peripherals */
 MX_GPIO_Init();
 MX_USART2_UART_Init();
 MX_QUADSPI_Init();
 MX_SPI1_Init();
 /* USER CODE BEGIN 2 */
 /* USER CODE END 2 */
 /* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
       {
           /* USER CODE BEGIN 3 */
           // 1. Send a message via USART1 and USART2
           sprintf((char*)buffer, "HELLO from STM32 via SPI Data\r\n");
           HAL_UART_Transmit(&huart2, buffer, strlen((char*)buffer), 1000);
// Send the message via USART2 (you can use USART1 if you want)
           HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_RESET);  // Pull CS
low
           if (HAL_SPI_TransmitReceive(&hspi1, txData, rxData,
sizeof(txData), HAL_MAX_DELAY) != HAL_OK) {
               // Handle error
           }
           HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_SET);  // Pull CS
high
           // 2. Send and receive data via SPI2 (Full-Duplex Communication)
           if (HAL_SPI_TransmitReceive(&hspi1, txData, rxData,
sizeof(txData), HAL_MAX_DELAY) != HAL_OK)
           {
               // SPI transmission error, handle it
               Error_Handler();
           }
           // 3. Debug: Print received SPI data via USART2
           for (int i = 0; i < sizeof(rxData); i++)
           {
               // Transmit received SPI data via USART2 for debugging
               HAL_UART_Transmit(&huart2, &rxData[i], 1, 1000);
           }
           // 4. Wait for 5 seconds before the next loop iteration
           HAL_Delay(5000);  // Delay for 5 seconds
           /* USER CODE END 3 */
       }
   }
/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
```

```
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 /** Configure the main internal regulator output voltage
 */
 __HAL_RCC_PWR_CLK_ENABLE();
 __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
 /** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLM = 16;
 RCC_OscInitStruct.PLL.PLLN = 336;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
 RCC_OscInitStruct.PLL.PLLQ = 2;
 RCC_OscInitStruct.PLL.PLLR = 2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
   Error_Handler();
 }
 /** Initializes the CPU, AHB and APB buses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                             |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
 {
   Error_Handler();
 }
}
/**
 * @brief QUADSPI Initialization Function
 * @param None
 * @retval None
 */
static void MX_QUADSPI_Init(void)
{
 /* USER CODE BEGIN QUADSPI_Init 0 */
 /* USER CODE END QUADSPI_Init 0 */
 /* USER CODE BEGIN QUADSPI_Init 1 */
 /* USER CODE END QUADSPI_Init 1 */
 /* QUADSPI parameter configuration*/
 hqspi.Instance = QUADSPI;
 hqspi.Init.ClockPrescaler = 255;
 hqspi.Init.FifoThreshold = 1;
 hqspi.Init.SampleShifting = QSPI_SAMPLE_SHIFTING_NONE;
```

```
 hqspi.Init.FlashSize = 1;
 hqspi.Init.ChipSelectHighTime = QSPI_CS_HIGH_TIME_1_CYCLE;
 hqspi.Init.ClockMode = QSPI_CLOCK_MODE_0;
 hqspi.Init.FlashID = QSPI_FLASH_ID_1;
 hqspi.Init.DualFlash = QSPI_DUALFLASH_DISABLE;
 if (HAL_QSPI_Init(&hqspi) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN QUADSPI_Init 2 */
 /* USER CODE END QUADSPI_Init 2 */
}
/**
 * @brief SPI1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI1_Init(void)
{
 /* USER CODE BEGIN SPI1_Init 0 */
 /* USER CODE END SPI1_Init 0 */
 /* USER CODE BEGIN SPI1_Init 1 */
 /* USER CODE END SPI1_Init 1 */
 /* SPI1 parameter configuration*/
 hspi1.Instance = SPI1;
 hspi1.Init.Mode = SPI_MODE_MASTER;
 hspi1.Init.Direction = SPI_DIRECTION_2LINES;
 hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
 hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
 hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
 hspi1.Init.NSS = SPI_NSS_SOFT;
 hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
 hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
 hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
 hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
 hspi1.Init.CRCPolynomial = 10;
 if (HAL_SPI_Init(&hspi1) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN SPI1_Init 2 */
 /* USER CODE END SPI1_Init 2 */
}
/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init(void)
{
 /* USER CODE BEGIN USART2_Init 0 */
 /* USER CODE END USART2_Init 0 */
 /* USER CODE BEGIN USART2_Init 1 */
```

```c
  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */
  /* USER CODE END USART2_Init 2 */
}
/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
/* USER CODE BEGIN 4 */
/* USER CODE END 4 */
/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
```

```c
void Error_Handler(void)
{
 /* USER CODE BEGIN Error_Handler_Debug */
 /* User can add his own implementation to report the HAL error return state
*/
 __disable_irq();
 while (1)
 {
 }
 /* USER CODE END Error_Handler_Debug */
}
#ifdef  USE_FULL_ASSERT
/**
 * @brief  Reports the name of the source file and the source line number
 *         where the assert_param error has occurred.
 * @param  file: pointer to the source file name
 * @param  line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
 /* USER CODE BEGIN 6 */
 /* User can add his own implementation to report the file name and line
number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line)
*/
 /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

Rugged Board Code :-

```python
import time

import sys

try:

import spidev
SPI_SUPPORTED = True


except ImportError:
        SPI_SUPPORTED = False


try:
import RPi.GPIO as GPIO
GPIO_SUPPORTED = True
except ImportError:
```

```
GPIO_SUPPORTED = False
# Define SPI Pins (for direct GPIO-based SPI)

MOSI_PIN = 3  # Pin for MOSI (Master Out Slave In)
MISO_PIN = 4  # Pin for MISO (Master In Slave Out)
SCK_PIN = 5   # Pin for SCK (Serial Clock)
CS_PIN = 6    # Pin for Chip Select (CS)



SPI_DEVICE = "/dev/spidev1.0"  # SPI bus and chip select (change this based on your
setup)

STRING_TO_SEND = "Rugged in Board SPI"  # The string data you want to send
MAX_RETRIES = 5  # Maximum number of retries in case of failure
SEND_INTERVAL = 5  # Send interval in seconds


# SPI Configuration
SPI_SPEED_HZ = 50000  # SPI speed (50 kHz)
SPI_MODE = 0  # SPI mode (CPOL=0, CPHA=0)
SPI_BITS_PER_WORD = 8  # 8 bits per word


def setup_spi_pins():


    """Setup SPI pins for GPIO-based SPI if GPIO is supported."""
if GPIO_SUPPORTED:
GPIO.setmode(GPIO.BCM)  # Use BCM pin numbering
GPIO.setup(MOSI_PIN, GPIO.OUT)  # Set MOSI as output
GPIO.setup(MISO_PIN, GPIO.IN)   # Set MISO as input
GPIO.setup(SCK_PIN, GPIO.OUT)   # Set SCK as output
GPIO.setup(CS_PIN, GPIO.OUT)     # Set CS as output

def open_spi_device():


    """Open SPI device using spidev if SPI is supported."""


    if SPI_SUPPORTED:


    try:

    spi = spidev.SpiDev()  # Create SPI object
    spi.open(0, 0)  # Open SPI bus 0, device 0 (this may vary depending on your setup)
    spi.max_speed_hz = SPI_SPEED_HZ  # Set SPI speed
    spi.mode = SPI_MODE  # Set SPI mode
```

```
        spi.bits_per_word = SPI_BITS_PER_WORD  # Set bits per word
        return spi
except IOError:

        print("Failed to open SPI device. Please check your connections.")

        sys.exit(1)

        else:


        return None

def SPI_SendData(spi, data):


        """Send data via SPI (Rugged Board as SPI Master)."""


        if not SPI_SUPPORTED:
print("SPI is not available, skipping data transmission.")


        return False



print(f"Sent data: '{data}'")  # Send the full string, not individual characters


        try:


        response = spi.xfer([ord(c) for c in data])  # Send each character as its byte value


        return True

        except IOError:


        print(f"Failed to write to the SPI bus while sending '{data}'")


        return False

      def SPI_ReceiveData(spi):
        """Receive data via SPI from STM32 (SPI Slave)."""
         retries = 0
```

```
while retries < MAX_RETRIES:

    try:

    # Send a dummy byte (0x00) to receive data from the slave (full-duplex
communication)

    if SPI_SUPPORTED:

        response = spi.xfer([0x00] * 16)  # Send 16 dummy bytes to receive a
response from STM32

        # Convert the response to text format

        received_text = ''.join([chr(byte) if 32 <= byte <= 126 else '.' for byte in
response])  # Handle non-printable characters

        print(f"Received: {received_text}")  # Display received data

        return received_text

    else:

 print("No SPI support for receiving data.")

        return None

    except IOError:

    retries += 1

    time.sleep(0.5)  # Wait before retrying

    print("Failed to read from the SPI bus after multiple attempts.")
    return None

   def main():

    """Main function to handle SPI communication between Rugged Board and
STM32."""

   setup_spi_pins()  # Initialize SPI pins (if GPIO supported)
spi = open_spi_device()  # Open the SPI device (if SPI is supported)
```

```
last_send_time = 0  # Track the last send time


  while True:

   current_time = time.time()

# Send data every SEND_INTERVAL seconds
if current_time - last_send_time >= SEND_INTERVAL:
if SPI_SendData(spi, STRING_TO_SEND):


last_send_time = current_time

 # Continuously receive data from STM32
received_data = SPI_ReceiveData(spi)

if received_data:

        print(f"Received: {received_data}")  # Display the received message


   # Small delay to avoid overwhelming the SPI bus
time.sleep(0.1)  # 100ms delay


 if __name__ == '__main__':


        try:
         main()
        except KeyboardInterrupt:
         print("Stopping SPI communication...")
        if GPIO_SUPPORTED:
         GPIO.cleanup()  # Clean up the GPIO settings when the script is interrupted (only if
GPIO is used)
```

Expecting output :-

Sent data: 'Rugged in Board SPI'

Received: `HELLO from STM32 via SPI Data`