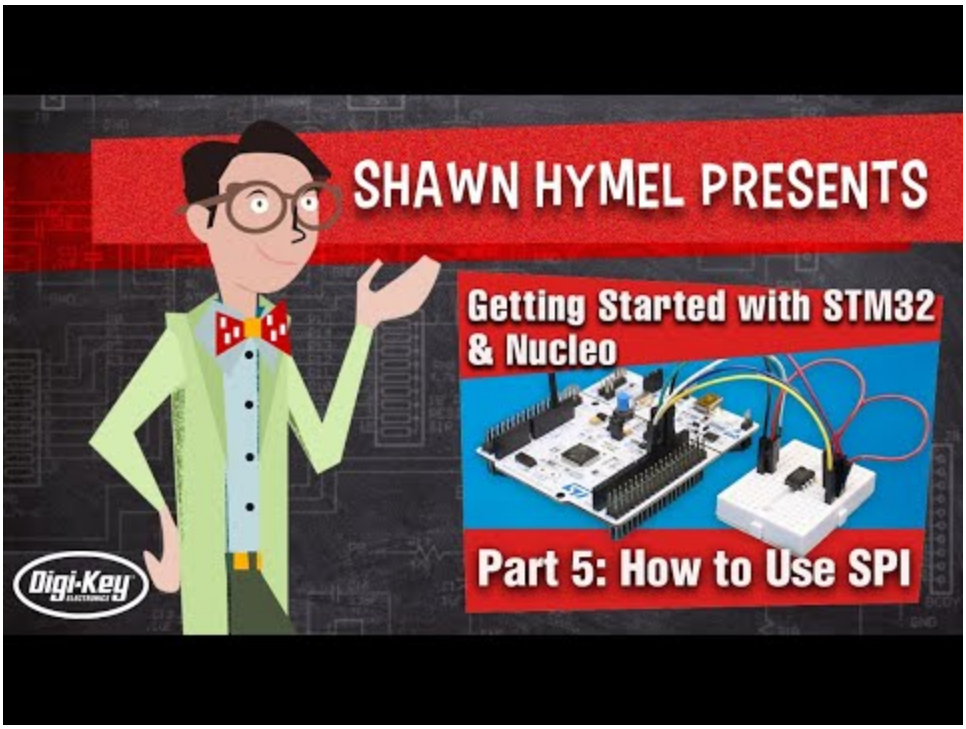
Getting Started with STM32 - How to Use SPI

ShawnHymel

By ShawnHymel

Like I2C, SPI is a common communication protocol in embedded devices. It often supports higher data rates than I2C, but it requires more pins to use. In this tutorial, I'll show you how to set up SPI using STM32CubeIDE and communicate with an external EEPROM chip.

You can watch this tutorial in video format here:



Watch Video At: https://youtu.be/eFKeNPJq50g

Please refer to this tutorial if you need a refresher on using STM32CubeIDE and working with GPIO in HAL.

Overview

Serial Peripheral Interface (SPI) was created by Motorola in the 1980s as a way to communicate between a microcontroller and various peripherals. It was never formalized into a standard, but other companies started using it soon after. As a result, it has become a *de facto* standard in the industry.

SPI is often used when a simple communication scheme is needed with faster speeds than what UART and I2C can provide. Because SPI is a synchronous communication scheme (there is a separate clock line), it can theoretically work up to any speed. However, you will often run into transmission line issues (cross talk, spurious signals, weak signals, etc.) when working above a few MHz. I find that speeds in the 1-10 MHz range work well for most of my needs (and work well enough on breadboards).

You will find many hardware peripheral devices that support SPI, including sensors like accelerometers, memory chips (e.g. EEPROM), and LED/LCD drivers.

SPI Wiring

SPI can be connected in a variety of configurations. The first, and probably most common, is the simple point-to-point connection scheme.

Note that because SPI is a de facto standard, you will come across different names for the lines:

- Clock: clock (CLK), serial clock (SCK)
- Controller data out: serial data out (SDO), master out slave in (MOSI), controller out peripheral in (COPI) (note that the SDO pin of the controller should be connected to the SDI pin of the peripheral)
- Controller data in: serial data in (SDI), master in slave out (MISO), controller in slave out (CIPO) (note that the SDI pin of the controller should be connected to the SDO pin of the controller)
- Select: chip select (CS), slave select (SS)

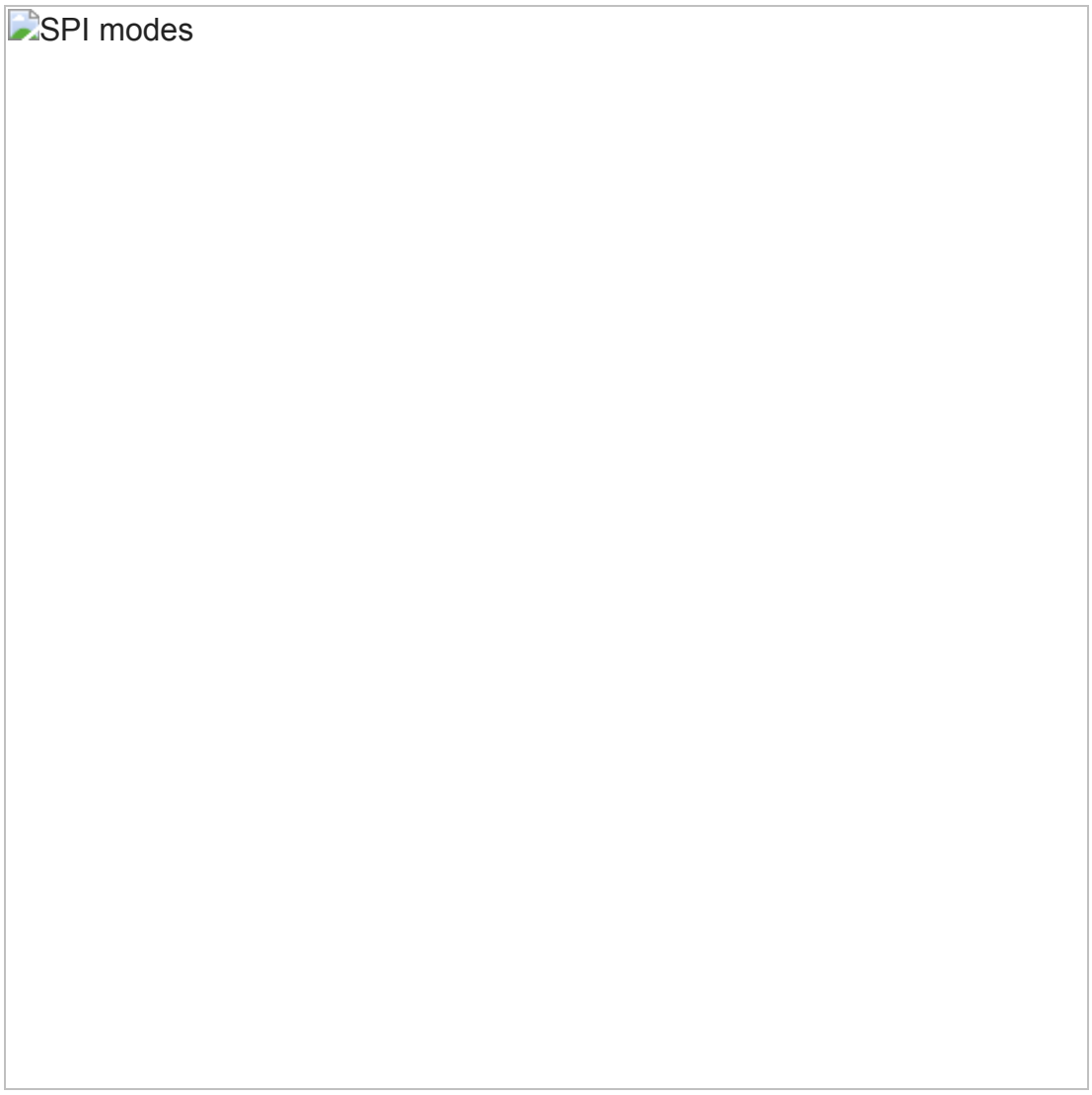
By default, SPI is full-duplex, which means you can transmit and receive at the same time. Sometimes, you will come across devices that only support half-duplex mode in order to save pins. The shared data line might be labeled as serial in/out (SI/O, SIO) or serial data in/out (SDIO). Note that for STM32 parts, the shared data line for half-duplex communication should be connected to the SDO (or MOSI) pin.

SPI is a bus, which means you can use multiple peripheral devices for one controller. All of the lines are shared except for the CS line. You will need to dedicate one pin on your controller as a separate CS line for each peripheral you wish to communicate with. Note that CS lines are active low, meaning they idle high, and we individually pull them low when we want to talk to a device.

Sometimes, you'll come across parts that work in a daisy chain configuration. This is popular with things like LED drivers. The SDO pin of the first peripheral should be connected to the SDI pin of the next peripheral in the chain. The final peripheral's SDO pin should be connected back to the controller.

SPI Modes

SPI has 4 different modes:



These modes refer to how data is sampled with the clock pulses. A clock polarity (CPOL) of 0 means that the clock line idles low whereas a CPOL of 1 means the clock line idles high. If clock phase (CPHA) is 0, bits are sampled on the leading clock edge and if CPHA is 1, bits are sampled on the trailing clock edge.

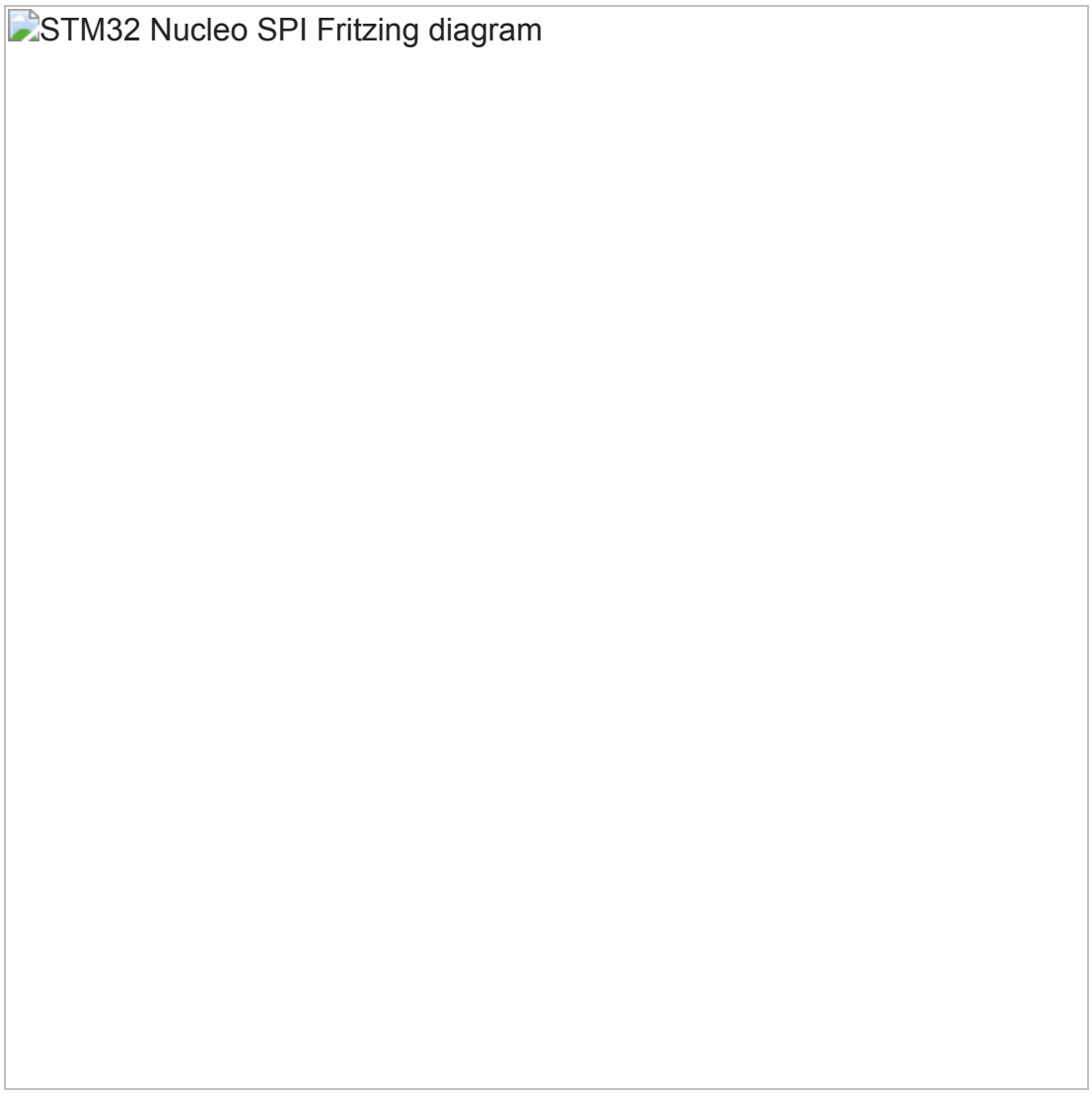
In my experience, most devices use SPI mode 0, which is CPOL = 0 and CPHA = 0.

SPI Timing

Below is a timing diagram I created using a real SPI transfer capture from my logic analyzer. To start communication, the controller pulls the CS line low and begins to pulse the clock. Normally, the MOSI and MISO lines can have data on them at the same time, but in this case, the EEPROM chip wants you to send a command first and then read second.

Hardware Hookup

Connect the EEPROM chip (I'm using a Microchip 25AA40A-I/P) to the Nucleo board as follows.



STM32CubeIDE Configuration

Start a new STM32 project, select your board (I'm using a Nucleo-L476RG), and give your project a memorable name. In the CubeMX tool, change the PA5 pin to **Reset_State** to disable it. This pin is connected to the LED on the Nucleo board. It's shared with the SPI SCK line, so we need to disable it before setting up SPI.

In *Connectivity*, select *SPI1*, and set *Mode* to **Full-Duplex Master**. Change *Data Size* to **8 bits** and change the *Prescaler* to **64** (we want the Baud Rate to be around 1 Mbits/s). Change *NSSP Mode* (slave-select pulse mode) to **Disabled**.

Note that we need to control pin D13 on our Nucleo as the chip select (CS) line manually. To do that, change pin *PB6* in the pinout view to **GPIO_Output**.

Save and open **main.c**.

Code

Enter the following code. Note that most of *main.c* will be auto-generated by CubeMX, so you only need to worry about the parts between the /* USER CODE */ guards.

Copy Code

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
SPI_HandleTypeDef hspi1;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

// 25AA040A instructions
const uint8_t EEPROM_READ = 0b00000011;
const uint8_t EEPROM_WRITE = 0b00000010;
const uint8_t EEPROM_WRDI = 0b00000100;
```

```
const uint8_t EEPROM_WREN = 0b00000110;
const uint8_t EEPROM_RDSR = 0b00000101;
const uint8_t EEPROM_WRSR = 0b00000001;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN0 */



/* USER CODE END0 */



/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN1 */
  char uart_buf[50];
  int uart_buf_len;
  char spi_buf[20];
  uint8_t addr;
  uint8_t wip;



  /* USER CODE END1 */



  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* 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_SPI1_Init();
MX_USART2_UART_Init();
/* USER CODE BEGIN2 */



// CS pin should default high
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Say something
uart_buf_len = sprintf(uart_buf, "SPI Test\r\n");
HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

// Enable write enable latch (allow write operations)
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_WREN, 1, 100);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Read status register
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_RDSR, 1, 100);
HAL_SPI_Receive(&hspi1, (uint8_t *)spi_buf, 1, 100);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Print out status register
uart_buf_len = sprintf(uart_buf,
                       "Status: 0xx\r\n",
                       (unsigned int)spi_buf[0]);
HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

// Test bytes to write to EEPROM
spi_buf[0] = 0xAB;
spi_buf[1] = 0xCD;
spi_buf[2] = 0xEF;

// Set starting address
addr = 0x05;

// Write 3 bytes starting at given address
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_WRITE, 1, 100);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&addr, 1, 100);
HAL_SPI_Transmit(&hspi1, (uint8_t *)spi_buf, 3, 100);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Clear buffer
spi_buf[0] = 0;
spi_buf[1] = 0;
```

```
spi_buf[2] = 0;

// Wait until WIP bit is cleared
wip = 1;
while (wip)
{
  // Read status register
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
  HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_RDSR, 1, 100);
  HAL_SPI_Receive(&hspi1, (uint8_t *)spi_buf, 1, 100);
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

  // Mask out WIP bit
  wip = spi_buf[0] & 0b00000001;
}

// Read the 3 bytes back
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_READ, 1, 100);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&addr, 1, 100);
HAL_SPI_Receive(&hspi1, (uint8_t *)spi_buf, 3, 100);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Print out bytes read
uart_buf_len = sprintf(uart_buf,
                       "0xx 0xx 0xx\r\n",
                       (unsigned int)spi_buf[0],
                       (unsigned int)spi_buf[1],
                       (unsigned int)spi_buf[2]);
HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

// Read status register
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_RDSR, 1, 100);
HAL_SPI_Receive(&hspi1, (uint8_t *)spi_buf, 1, 100);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

// Print out status register
uart_buf_len = sprintf(uart_buf,
                       "Status: 0xx\r\n",
                       (unsigned int)spi_buf[0]);
HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

/* USER CODE END2 */



/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
```

```
    /* USER CODE BEGIN3 */
  }
  /* USER CODE END3 */
}

  /***
 * REST OF MAIN.C
 * ...
 */
```

Run your code and connect to your Nucleo board using a serial program, such as PuTTY. You should see our start message, “SPI Test” being printed out, followed by the contents of the STATUS register in the EEPROM (0x02) after we set the WEL bit. Then, you should see the 3 bytes we wrote to the EEPROM being read back (0xab 0xcd 0xef). Finally, you should see the STATUS register value again (0x00) after the read command completes.

Using SPI in Interrupt Mode

Most STM32 chips also support using SPI in interrupt mode. This allows you to make non-blocking code that handles transmitting and receiving in the background. I've created an example of a non-blocking SPI transmitter/receiver for you to use as a starting point.

Note that you will need to go back into the CubeMX perspective and enable the *SPI1 global interrupt* option in *Connectivity > SPI1 > NVIC Settings*.

Back in *main.c*, change the code to the following:

Copy Code

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
SPI_HandleTypeDef hspi1;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

// 25AA040A instructions
const uint8_t EEPROM_READ = 0b00000011;
const uint8_t EEPROM_WRITE = 0b00000010;
const uint8_t EEPROM_WRDI = 0b00000100;
```

```
const uint8_t EEPROM_WREN = 0b00000110;
const uint8_t EEPROM_RDSR = 0b00000101;
const uint8_t EEPROM_WRSR = 0b00000001;

// Global flags
volatile uint8_t spi_xmit_flag = 0;
volatile uint8_t spi_recv_flag = 0;

/* 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_SPI1_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN0 */



/* USER CODE END0 */



/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN1 */
  char uart_buf[50];
  int uart_buf_len;
  char spi_buf[20];
  uint8_t addr;
  uint8_t wip;
  uint8_t state = 0;
  /* USER CODE END1 */



  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* 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_SPI1_Init();
  /* USER CODE BEGIN2 */



  // CS pin should default high
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

  // Say something
  uart_buf_len = sprintf(uart_buf, "SPI Interrupt Test\r\n");
  HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

  // Set starting address in EEPROM (arbitrarily set to 5). Note that for the
  // 25AA040A, we can't do sequential writes outside of page (16 bytes)
  addr = 0x05;

  /* USER CODE END2 */



  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    // Finite state machine to allow for non-blocking SPI transmit/receive
    switch(state)
    {
// Transmit
case0:



// First 2 bytes of buffer are instruction and address
        spi_buf[0] = EEPROM_WRITE;
        spi_buf[1] = addr;



// Fill buffer with stuff to write to EEPROM
for (int i = 0; i < 10; i++)
        {
          spi_buf[2 + i] = i;
        }
```

```
  // Enable write enable latch (allow write operations)
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
  HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_WREN, 1, 100);
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

  // Perform non-blocking write to SPI
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
  HAL_SPI_Transmit_IT(&hspi1, (uint8_t *)spi_buf, 12);

  // Go to next state: waiting for interrupt flag
  state += 1;

  break;

// Wait for transmit flag
case 1:

  if (spi_xmit_flag)
  {
    // Clear flag and go to next state
    spi_xmit_flag = 0;
    state += 1;
  }

  break;

// Wait for WIP bit to be cleared
case 2:

  // Read status register
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
  HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_RDSR, 1, 100);
  HAL_SPI_Receive(&hspi1, (uint8_t *)spi_buf, 1, 100);
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);

  // Mask out WIP bit
  wip = spi_buf[0] & 0b00000001;

  // If WIP is cleared, go to next state
  if (wip == 0)
  {
    state += 1;
  }

  break;

// Set up for interrupt-based SPI receive
case 3:

  // Clear SPI buffer
  for (int i = 0; i < 12; i++)
  {
    spi_buf[i] = 0;
  }
```

```
      // Read the 10 bytes back
      HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
      HAL_SPI_Transmit(&hspi1, (uint8_t *)&EEPROM_READ, 1, 100);
      HAL_SPI_Transmit(&hspi1, (uint8_t *)&addr, 1, 100);
      HAL_SPI_Receive_IT(&hspi1, (uint8_t *)spi_buf, 10);

      // Go to next state: waiting for receive to finish
      state += 1;

      break;

    // Wait for receive flag
    case 4:

      if (spi_recv_flag)
      {
        // Clear flag and go to next state
        spi_recv_flag = 0;
        state += 1;
      }

      break;

    // Print out received bytes and wait before retransmitting
    case 5:

      // Print out bytes
      for (int i = 0; i < 10; i++)
      {
        uart_buf_len = sprintf(uart_buf,
                               "0xx ",(unsigned int)spi_buf[i]);
        HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);
      }

      // Print newline
      uart_buf_len = sprintf(uart_buf, "\r\n");
      HAL_UART_Transmit(&huart2, (uint8_t *)uart_buf, uart_buf_len, 100);

      // Wait a few seconds before retransmitting (yes, I know that this is
      // blocking--you can make it non-blocking if you wish. I'm lazy.)
      HAL_Delay(1000);
      state = 0;

      break;

    default:
      break;
  }

  /* USER CODE END WHILE */

  /* USER CODE BEGIN3 */
}
/* USER CODE END3 */
```

```
}

  /***
 * AUTO-GENERATED FUNCTIONS
 * ...
 */
```

We need to add our interrupt handlers, which reset the CS pin and raise the transmit or receive flag for our state machine. Find the /* USER CODE BEGIN 4 */ guard and enter the following:

Copy Code

```
/* USER CODE BEGIN4 */



// This is called when SPI transmit is done
void HAL_SPI_TxCpltCallback (SPI_HandleTypeDef * hspi)
{
  // Set CS pin to high and raise flag
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
  spi_xmit_flag = 1;
}

// This is called when SPI receive is done
void HAL_SPI_RxCpltCallback (SPI_HandleTypeDef * hspi)
{
  // Set CS pin to high and raise flag
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
  spi_recv_flag = 1;
}

/* USER CODE END 4 */
```

Note that I use a state machine to handle transmitting and receiving. You can write code after the switch/case statement, which should run while the microcontroller handles sending and receiving SPI data in the background.

Resources and Going Further

One thing I did not cover was using SPI with DMA. This is possible for many STM32 parts, but you will need to enable DMA requests as shown in this tutorial (it covers DMA with ADC, but the steps should be similar).

The following documents are available if you want to learn more about SPI on the STM32:

- Nucleo-L476RG Pinout
- STM32L4 HAL API Documentation

- Microchip 25AA040 EEPROM Datasheet

Recommended Reading

- Getting Started with STM32 and Nucleo Part 1: Introduction to STM32CubeIDE and Blinky
- Getting Started with STM32 and Nucleo Part 2: How to Use I2C to Read Temperature Sensor TMP102
- Getting Started with STM32 and Nucleo Part 3: How to Run Multiple Threads with CMSIS-RTOS Interface
- Getting Started With STM32 & Nucleo Part 4: Working with ADC and DMA
- Getting Started with STM32 - How to Use SPI

Key Parts and Components

2 Items

Add all Digi-Key Parts to Cart

Have questions or comments? Continue the conversation on TechForum, DigiKey's online community and technical resource.

TechForum

Visit TechForum
Arduino | 3D Printing | Raspberry Pi