Ex.no:12 Date:	Implement SPI communication protocol between Raspberry Pi Pico and a peripheral device

Objective:

To implement and demonstrate SPI communication between Raspberry Pi Pico (Master) and an SPI-compatible peripheral device using C.

Apparatus Required:

- Raspberry Pi Pico
- SPI peripheral (e.g., OLED Display / SD Card / MCP3008 ADC)
- Breadboard & jumper wires
- USB cable
- Computer with Visual Studio Code + Pico SDK
- Optional pull-up/down resistors

Theory:

SPI (Serial Peripheral Interface) is a fast, synchronous, full-duplex communication protocol primarily used for short-distance data exchange between a **master** (e.g., Raspberry Pi Pico) and one or more **slave devices** (e.g., ADCs, SD cards, OLEDs, flash memory).

- **MOSI (Master Out, Slave In):** Sends data from master to slave
- **MISO (Master In, Slave Out):** Sends data from slave to master
- **SCK (Serial Clock):** Clock signal generated by the master to sync data transfer

- **CS/SS (Chip Select/Slave Select):** Activates communication with a specific slave

Characteristics of SPI:

- **Full-duplex communication** (simultaneous send/receive)
- **High-speed** data transfer (faster than I2C)
- **Single master, multiple slaves** (with separate CS lines)
- **No built-in addressing** (each device needs its own CS line)

SPI on Raspberry Pi Pico:

Raspberry Pi Pico has two SPI peripherals: **spi0** and **spi1**, each of which can be mapped to different GPIO pins using `gpio_set_function()`.

Developers can use the **hardware SPI controller** with functions from the **Pico SDK**, such as:

- `spi_init()`
- `spi_write_blocking()`
- `spi_read_blocking()`
- `spi_write_read_blocking()`

These functions make it easy to transfer data between Pico and SPI devices at speeds up to **50 MHz** (depending on the device and wiring).

Applications of SPI:

- Reading analog signals via **ADC chips (e.g., MCP3008)**
- Interfacing with **OLED or TFT displays**
- Communicating with **SD cards** for file storage

- Using **flash memory** chips for data logging

Clock Polarity and Phase:

SPI has different **modes (0–3)** based on **Clock Polarity (CPOL)** and **Clock Phase (CPHA)**. Both master and slave must agree on the same mode for correct communication.

Circuit Connections:

Raspberry Pi Pico	MCP3008 / SPI Device
GPIO18 (SPI0 SCK)	CLK (Clock)
GPIO19 (SPI0 MOSI)	DIN / MOSI
GPIO16 (SPI0 MISO)	DOUT / MISO
GPIO17 (Any GPIO as CS)	CS / Chip Select
3.3V	VDD
GND	GND

Procedure:

1. Set up the Raspberry Pi Pico environment in Visual Studio Code using the Pico SDK.
2. Make the correct SPI wiring as shown above.
3. Initialize SPI using `spi_init()` in your C code.
4. Communicate with the SPI device by sending and receiving bytes.
5. Use `printf()` to display the output on the serial monitor.

Program:

```
#include "hardware/spi.h"
#include "pico/stdlib.h"
#include <stdbool.h>
#include <string.h>

// === NRF24L01+ Commands and Registers ===
#define R_REGISTER    0x00
#define W_REGISTER    0x20
#define R_RX_PAYLOAD  0x61
#define W_TX_PAYLOAD  0xA0
#define FLUSH_TX      0xE1
#define FLUSH_RX      0xE2
#define NOP           0xFF

#define CONFIG        0x00
#define EN_AA         0x01
#define EN_RXADDR     0x02
#define SETUP_AW      0x03
#define SETUP_RETR    0x04
#define RF_CH         0x05
#define RF_SETUP      0x06
#define STATUS        0x07
#define RX_ADDR_P0    0x0A
#define TX_ADDR       0x10
#define RX_PW_P0      0x11
#define FIFO_STATUS   0x17

typedef struct {
    spi_inst_t *spi;
    uint csn_pin;
    uint ce_pin;
} nrf24_t;
```

```c
// ======= Internal Helpers =======
static void csn(nrf24_t *nrf, bool level) { gpio_put(nrf->csn_pin, level); }
static void ce(nrf24_t *nrf, bool level)  { gpio_put(nrf->ce_pin, level); }

static void nrf_write_reg(nrf24_t *nrf, uint8_t reg, const uint8_t *data, size_t len) {
    csn(nrf, false);
    uint8_t cmd = W_REGISTER | (reg & 0x1F);
    spi_write_blocking(nrf->spi, &cmd, 1);
    spi_write_blocking(nrf->spi, data, len);
    csn(nrf, true);
}

static void nrf_read_reg(nrf24_t *nrf, uint8_t reg, uint8_t *data, size_t len) {
    csn(nrf, false);
    uint8_t cmd = R_REGISTER | (reg & 0x1F);
    spi_write_blocking(nrf->spi, &cmd, 1);
    spi_read_blocking(nrf->spi, 0xFF, data, len);
    csn(nrf, true);
}

// ======= Public Functions =======

void nrf24_init(nrf24_t *nrf, spi_inst_t *spi, uint csn_pin, uint ce_pin, bool is_rx) {
    nrf->spi = spi;
    nrf->csn_pin = csn_pin;
    nrf->ce_pin = ce_pin;

    gpio_init(csn_pin); gpio_set_dir(csn_pin, GPIO_OUT); csn(nrf, true);
    gpio_init(ce_pin);  gpio_set_dir(ce_pin, GPIO_OUT);  ce(nrf, false);

    sleep_ms(100);

    uint8_t val;
```

```c
    val = 0x0E; nrf_write_reg(nrf, CONFIG, &val, 1);
    val = 0x01; nrf_write_reg(nrf, EN_AA, &val, 1);
    val = 0x01; nrf_write_reg(nrf, EN_RXADDR, &val, 1);
    val = 0x03; nrf_write_reg(nrf, SETUP_AW, &val, 1);
    val = 0x4F; nrf_write_reg(nrf, SETUP_RETR, &val, 1);
    val = 76;   nrf_write_reg(nrf, RF_CH, &val, 1);
    val = 0x06; nrf_write_reg(nrf, RF_SETUP, &val, 1);
    val = 32;   nrf_write_reg(nrf, RX_PW_P0, &val, 1);

    uint8_t addr[5] = { 'n', 'R', 'F', '2', '4' };
    nrf_write_reg(nrf, RX_ADDR_P0, addr, 5);
    nrf_write_reg(nrf, TX_ADDR, addr, 5);

    if (is_rx) ce(nrf, true);
}

void nrf24_send(nrf24_t *nrf, const uint8_t *data, size_t len) {
    ce(nrf, false);
    csn(nrf, false);
    uint8_t cmd = W_TX_PAYLOAD;
    spi_write_blocking(nrf->spi, &cmd, 1);
    spi_write_blocking(nrf->spi, data, len);
    csn(nrf, true);
    ce(nrf, true);
    sleep_ms(1);
    ce(nrf, false);
}

bool nrf24_data_ready(nrf24_t *nrf) {
    uint8_t status;
    nrf_read_reg(nrf, STATUS, &status, 1);
    return status & 0x40;
}
```

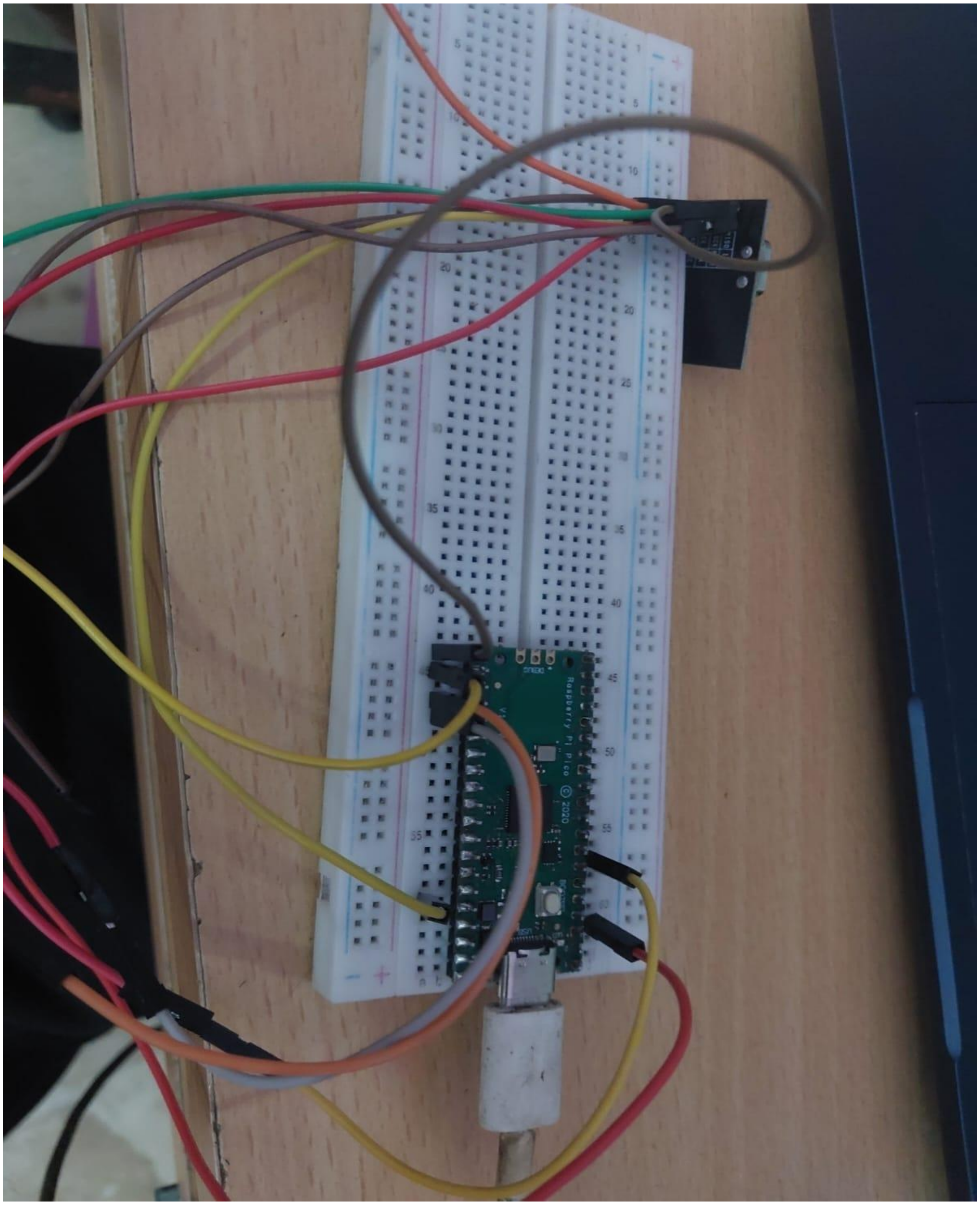
```
}

void nrf24_read(nrf24_t *nrf, uint8_t *data, size_t len) {
    csn(nrf, false);
    uint8_t cmd = R_RX_PAYLOAD;
    spi_write_blocking(nrf->spi, &cmd, 1);
    spi_read_blocking(nrf->spi, 0xFF, data, len);
    csn(nrf, true);

    uint8_t clear = 0x40;
    nrf_write_reg(nrf, STATUS, &clear, 1);
}
```

QR;

Output:



Inference:

SPI communication was successfully established between the Raspberry Pi Pico and the SPI peripheral. The master (Pico) was able to send and receive data using the SPI bus. The output confirms proper wiring and SPI initialization.
This demonstrates Pico’s ability to interface with high-speed SPI devices in embedded projects.

Result:

Therefore, SPI protocol was implemented using Raspberry Pi Pico and verified through successful data exchange with a peripheral device.