



Lab5: RS232, I2C and SPI Interfaces

Name : : Dhrubo Jouti Das Troyee

Student ID : 22663281

Name of the Unit : Transmission and Interface Design 203

Submission Date : 23 May 2025

Due Date : 23 May 2025

Lab Day(s) and Time : Friday 3-5pm

Lab Supervisor(s) : Mr.King Su Chan

DECLARATION

I hereby declare this is entirely my own work except for the references quoted. I declare that I have read the University's statement on plagiarism and copyright protection and that I have upheld them in producing this work.

A rectangular box containing a handwritten signature in black ink. The signature appears to read "TROYE".

SIGNATURE

Table of Contents

Serial no	Topics	Page number
1	Abstract	3
2	Aim	3
3	Introduction	3
4	Background	4
5	Equipment	4
6	Procedure	5-7
7	Observation	8-9
8.	Result	10
9	Discussion	11-13
10	Conclusion	14
11	References	14
12	Appendix	15

Abstract

In this report, the use of Arduino Uno boards to implement three popular communication protocols (RS232, I²C, and SPI) is examined. Both single-slave and multi-slave variations of the I²C interface were evaluated. Data was transferred via RS232 from a PC to an Arduino, which then relayed it via I²C. Two Arduinos were used to deliver serial data under different clock settings to verify SPI communication. Protocol-specific behaviour in speed, data handling, and accuracy was highlighted by the results. The experiment offered useful information about how serial communication protocols interact and function in embedded devices.

Aims

In this experiment, Arduino boards ((Master, Slave) were used to investigate and examine the behaviour and functioning of RS232, I²C, and SPI interfaces. The goal was to use Arduino boards and standard interface methods to enable serial communication between a microcontroller and a PC as well as between several microcontrollers.

Introduction

In embedded systems, serial communication is essential for facilitating smooth data transfer between sensors, controllers, and host PCs. The most widely used protocols, RS232, I²C, and SPI, are each developed to meet specific requirements.

Although it requires more complex wiring, SPI allows full-duplex, high-speed communication over four lines, making it appropriate for quicker applications. I²C allows for address-based data transmission during communication with several slave devices via two wires. A common point-to-point protocol for PC-to-device communication is RS232.

In this experiment, Arduino Uno microcontrollers are used to investigate the use of various protocols. The configurations include SPI communication at various clock speeds, RS232 data relaying to I²C, and I²C communication in single- and multi-slave configurations.

Background

The efficient and reliable interchange of data between devices is defined by communication protocols in embedded systems. In addition to control signals, RS232, one of the first serial protocols, uses different lines for sending and receiving data. Despite being straightforward and trustworthy it is a bit slow and only allows point-to-point communication. But because of its low data rates and one-to-one communication support, it is less appropriate for contemporary high-speed applications. Some of the common application of RS232 can be seen in industrial area (CNC machines, Industrial Control system) , networking area (Configuring routers, switch), medical Devices.

Higher data rates are offered via SPI (Serial Peripheral Interface), which uses four separate lines: MISO(Master In Slave Out), MOSI(Master Out Slave In), SCK(Clock), and SS (Slave clock). It doesn't need addressing and can communicate in full duplex, in contrast to I²C, but it doesn't handle many devices properly except it is controlled externally.SD card Readers, LCD Screens, memory Modules, Audio Devices are some of the applications of SPI.

The purpose of I²C (Inter-Integrated Circuit) was to simplify wiring. It supports one master and several slaves across two bidirectional lines, SDA for data and SCL for clock. Unique addressing is necessary for communication, which makes it ideal for peripheral control and sensor networks. Because of its effective multi-device handling, I²C is commonly used in applications including digital sensors , analog sensors ,temperature sensors, display modules (character display), memory expansion and real-time clocks.

Equipment

1. Digital Storage oscilloscope (DSO) and probes
2. 3 Arduino boards and USB Cables
3. Matrix board
4. 2 10k Ω resistors
5. 2 200 Ω resistors
6. 3 LED
7. Wire cutters and Wires

Procedure

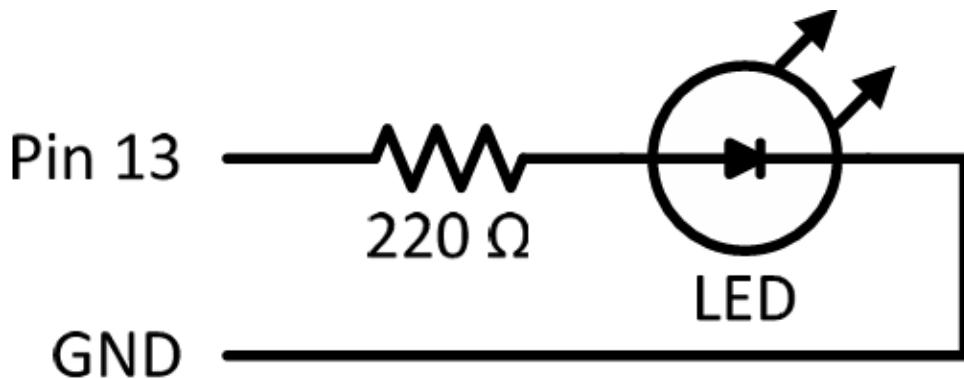


Figure 1: LED connected to Arduino pin 13

➤ **I²C Interface:**

1. Connecting analog pin4 (SDA-Slave Data) of the master to analog pin 4 of the slave.
2. Connecting analog pin5 (SCL-Slave Clock) of the master to analog pin 5 of the slave.
3. Connecting SDA(A4) and SCL(A5) of both boards to a 5v supply via a 10k Ω resistors.
4. Connecting a LED in series with a 200 Ω resistors between pin 13 and ground of both boards as shown in the figure 1 above.
5. Connecting the Arduino boards to the PC using USB cables. Now, uploading the program “Blink_I2C_Master.ino” to Master Board and “blink_I2C_slave.ino” to slave board . Observing the operating of both boards.
6. Now disconnecting the SCL (Analog pin 5) from Slave and observing again.
7. Reconnecting SCL(analog 5) and disconnecting SDA (analog4) and observing the operation.
8. Reconnecting Both SDA and SCL and connecting 2 DSO probes to SDA (analog 4) and SCL (Analog 5). After that, observing the signal patterns.

9. Changing the I²C address of slave device and observing the changes.
10. Connecting 2 Arduino boards as slave device to 1 master device. Assigning addresses 2 to one slave device and address 4 to another slave device. Uploading the program “Blink_I2C_master_2.ino” to the master device. Now observing the operation.

➤ **Communicate using Multiple Standards: RS232 (Serial) to I²C**

1. Connecting analog pin4 (SDA-Slave Data) of the master to analog pin 4 of the slave.
2. Connecting analog pin5 (SCL-Slave Clock) of the master to analog pin 5 of the slave.
3. Connecting SDA(A4) and SCL(A5) of both boards to a 5v supply via a 10k Ω resistors.
4. Connecting the two Arduino boards to the PC using USB cables.
5. Now, uploading the program “Serial_I2C_Master.ino” to Master Board and “Serial_I2C_slave.ino” to slave board.
6. Opening the serial Monitor on both computers.
7. Typing something on the serial monitor of the master and sending it.
8. Observing the operation of both boards.
9. Introducing a 500 ms delay at the end of the master’s loop () function by adding delay (500).
10. Saving and uploading the new version to master Arduino.
11. Observing the difference as well as operation.

➤ **SPI Interface:**

Master	Slave
SCK (Pin 13)	SCK (Pin 13)
MISO (Pin 12)	MISO (Pin 12)
MOSI (Pin 11)	MOSI (Pin 11)
SS (Pin 10)	SS (Pin 10)
GND	GND

Figure 2: Pin connection between Master and Slave Arduino Board

1. Connecting the two Arduino Board based on the figure.
2. Connecting the two Arduino boards to the PC using USB cables.
3. Now, uploading the program “SPI_Master.ino” to Master Board and “SPI_slave.ino” to slave board.
4. Opening the serial Monitor on both computers.
5. Typing something on the serial monitor of the master and sending it.
6. Observing the operation.
7. Replacing the parameter “SPI_CLOCK_DIV32” in master code with “SPI_CLOCK_DIV2” and observing the changes.
8. Again, replacing the parameter “SPI_CLOCK_DIV2” in master code with “SPI_CLOCK_DIV8” and observing the changes.

Observations

➤ **I²C Interface:**

- ❖ **uploading the program “Blink_I2C_Master.ino” to Master Board and “blink_I2C_slave.ino” to slave board:**

The Master and Slave boards' LED blink together.

- ❖ **Disconnecting SCL:**

The slave's LED turned on without blinking while the master continued to blink without making any difference.

- ❖ **Disconnecting SDA:**

The slave's LED turned off, but the master device continued to blink without making any changes.

- ❖ **Changing the I²C address of slave device :**

In this case, the LED of one slave and master blinked simultaneously and then the second slave also repeat the same thing, indicating each slave communicating to master in a specific order.

- ❖ **Assigning addresses 2 to one slave device and address 4 to another slave device(“Blink_I2C_Master_2.ino”):**

Slave LED stopped blinking and turned off, which indicates that the master did not recognize the new address.

➤ **Communicate using Multiple Standards: RS232 (Serial) to I²C**

- ❖ **Uploading the program “Serial_I2C_Master.ino” to Master Board and “Serial_I2C_slave.ino” to slave board and opening the serial monitor:**

When a message typed on the serial monitor of the master it instantly transmitted to the slave device.

- ❖ **Introducing a 500 ms delay at the end of the master’s loop () function by adding delay (500):**

The slave is still receiving messages, but the frequency of message transmission significantly decreases, with each character appearing one at a time after delay.

➤ **SPI Interface:**

❖ **The parameter “SPI_CLOCK_DIV32” in master code:**

In this case, Slave received the message slowly , with characters appearing one by one on the serial monitor,

❖ **Replacing the parameter “SPI_CLOCK_DIV32” in master code with “SPI_CLOCK_DIV2” and observing the changes:**

Slave received the message instantly, but the output appeared horizontally without proper formatting also contain unreadable characters like question marks and symbols.

❖ **Again, replacing the parameter “SPI_CLOCK_DIV2” in master code with “SPI_CLOCK_DIV8” and observing the changes:**

In this case, the message appeared on the slave's Serial Monitor quickly and without delay. Moreover, the displayed message was correct formatting, readable and clear.

Result

The comparison of SPI communication performance using two different clock divider settings: “**SPI_CLOCK_DIV2**” and “**SPI_CLOCK_DIV8**” are given below:

parameters	“ SPI_CLOCK_DIV8 ”	“ SPI_CLOCK_DIV2 ”
Clock frequency	2MHz	8MHz
Slave Output	Correct data	Unreadable/corrupted data
Slave Performance	Enough time to process data	unable to keep up with speed
Stability	Stable	Unstable
Communication result	Unreliable	Reliable
Data Integrity	Low possibility of errors	High possibility of data loss
Transmission speed	Moderate speed	Higher speed

*Figure 3: comparison between “**SPI_CLOCK_DIV2**” and “**SPI_CLOCK_DIV8**”*

Discussion



Figure 4: Waveform of I²C signal for SDA and SCL

The above waveform of the oscilloscope the green mark indicates the SCL (clock) line, and the yellow signal indicates the SDA (data) line. Unfortunately, the signals appear more like distorted sine waves rather than clean square pulses. This is probably because of the unwanted electrical effects that are generated by using various connection points, breadboards, and long jumper wires. The signal integrity in the multi-device communication system was damaged by these electrical effects

In this examination, both SDA and SCL signals were expected to display clear, square pulses with synchronous transitions, representing the I²C bus behaves during communication. However, the lab supervisor confirmed that the configuration was accurate.

Communication between the Arduino boards was constant and operating despite the waveform distortion, indicating that the I²C protocol is capable of handling mechanical flaws.

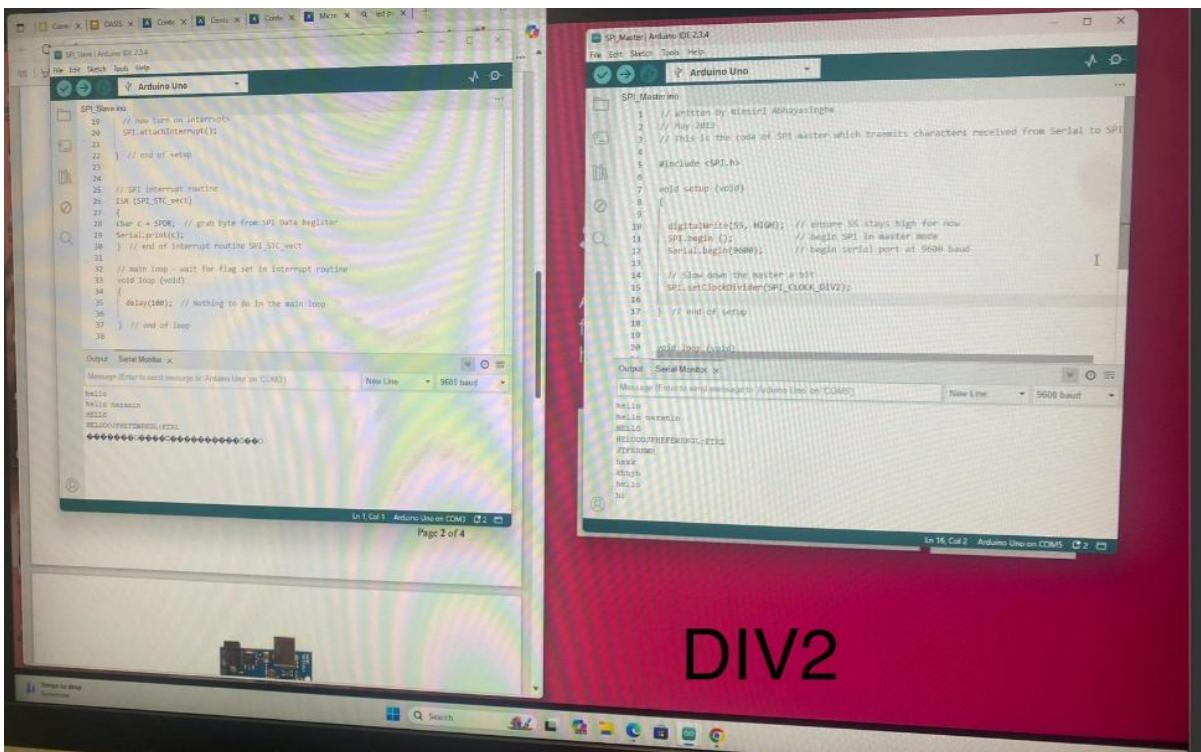


Figure 5: Snapshots of using “SPI_CLOCK_DIV2” parameter

According to Figure 4, the SPI clock speed is set to 8 MHz, the fastest supported on an Arduino Uno with a 16 MHz system clock, by specifying the SPI master (right side) with SPI_CLOCK_DIV2. Incoming data is received via an interrupt by the slave device (left side), which then delivers it to the serial monitor. The slave output shows unreadable, corrupted characters as well as missing or symbol replaced data even when the master delivers legitimate messages like “hello,” “HELLO,” and “hello naznin” This shows that the high-speed data stream is too fast for the slave to handle.

It is believed that the timing issue occurs as the slave is unable to read every byte from the SPI Data Register or finish the Serial.print() function before next byte arrives. To ensure reliable and accurate communication, it is crucial to consider the slave's processing limitations as well as the hardware setup quality, even if raising the SPI frequency to 8 MHz can increase data transfer rates.

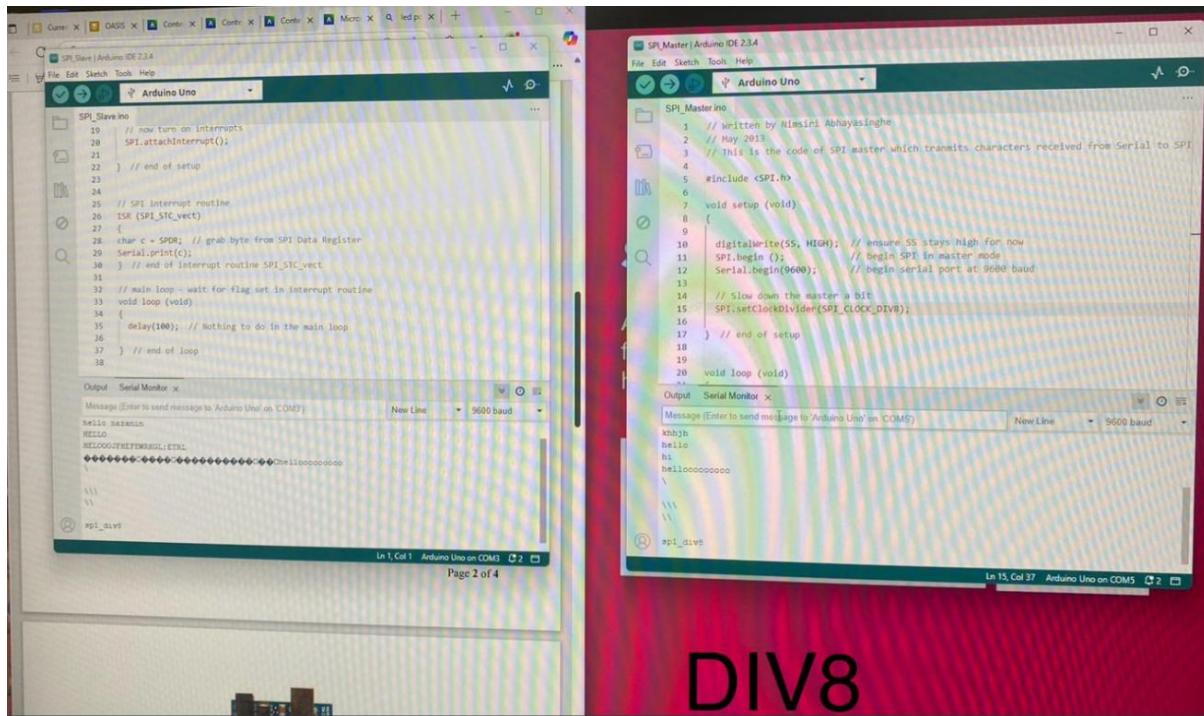


Figure 6: Snapshots of using “`SPI_CLOCK_DIV8`” parameter

This setup reduces the SPI clock speed on an Arduino Uno to 2 MHz (16 MHz / 8) by setting the SPI master (right side) using `SPI_CLOCK_DIV8`. Data is received by the slave device (left side) through an interrupt, and the characters appear to the serial monitor. It is evident from the picture that the slave correctly receives and displays almost all the master's messages, including "hello," "HELLOOOO" on the serial monitor. In comparison to `SPI_CLOCK_DIV2`, the transmission is generally far more consistent, even though there are still a few unreadable or garbled characters displayed.

This suggests that the slave has enough time to read every byte from the SPI Data Register and run `Serial.print()` without becoming overloaded when the SPI clock reduces to 2 MHz. The slave can accurately receive and transmit data with more time to process each byte, which reduces transmission errors and improves connection stability.

Conclusions

In summary, this project effectively illustrated how to use Arduino Uno boards to implement and function with RS232, I²C, and SPI communication protocols. Regarding the speed, structure, and reliability of data transport, each protocol had unique features. I²C made use of address-based control to facilitate communication with several devices, though hardware setup impacted the signal quality. RS232 was reliable for simple transmission highlighting the importance of software delays in message timing. Though signal distortion and data corruption were noted at high clock speeds, SPI offered the fastest data transport. All communication configurations worked exactly as expected, despite slight waveform changes because of practical setup.

The exercise gave participants a realistic understanding of the trade-offs and practical considerations that go into choosing and implementing serial communication protocols in embedded systems. Overall, this experiment provided a fundamental understanding of how to effectively develop and analyse serial communication strategies in an embedded real-world context.

References

- [1]“What is I2C: A No-Nonsense Explanation for You Right Here...,” *Best Microcontroller Projects*, 2025. <https://www.best-microcontroller-projects.com/what-is-i2c.html> (accessed May 25, 2025).
- [2]Spiceworks.com, 2024. <https://www.spiceworks.com/tech/networking/articles/what-is-spi/>
- [3]“Basics of Serial Peripheral Interface (SPI),” *Electronics Hub*, Jun. 20, 2017. <https://www.electronicshub.org/basics-serial-peripheral-interface-spi/>
- [4]Technical Editor, “Applications of serial peripheral interface (SPI) - Polytechnic Hub,” *Polytechnic Hub*, May 02, 2017. <https://www.polytechnichub.com/applications-serial-peripheral-interface-spi/> (accessed May 25, 2025).
- [5]D. Workshop, “I2C Communications Part 1 - Arduino to Arduino,” *DroneBot Workshop*, Mar. 30, 2019. <https://dronebotworkshop.com/i2c-arduino-arduino/>
- [6]R. Teja, “Arduino I2C Tutorial | How to use I2C Communication on Arduino?,” *ElectronicsHub*, Jan. 29, 2021. <https://www.electronicshub.org/arduino-i2c-tutorial/>

Appendix

Nil