Embedded Systems CSEN701

Dr. Catherine Elias

Eng. Abdalla Mohamed

Office: C1.211
Mail : abdalla.abdalla@guc.edu.eg

Eng. Mohamed Elshafie

Office: C1.211
Mail : Mohamed.el-shafei@guc.edu.eg

Eng. Maysarah El Tamalawy

Office: C7.201
Mail : Maysarah.mohamed@guc.edu.eg

Outline :

- **Introduction to communication**
- Serial communication
- UART protocol
- UART example
- I2C protocol
- I2C Example

communication

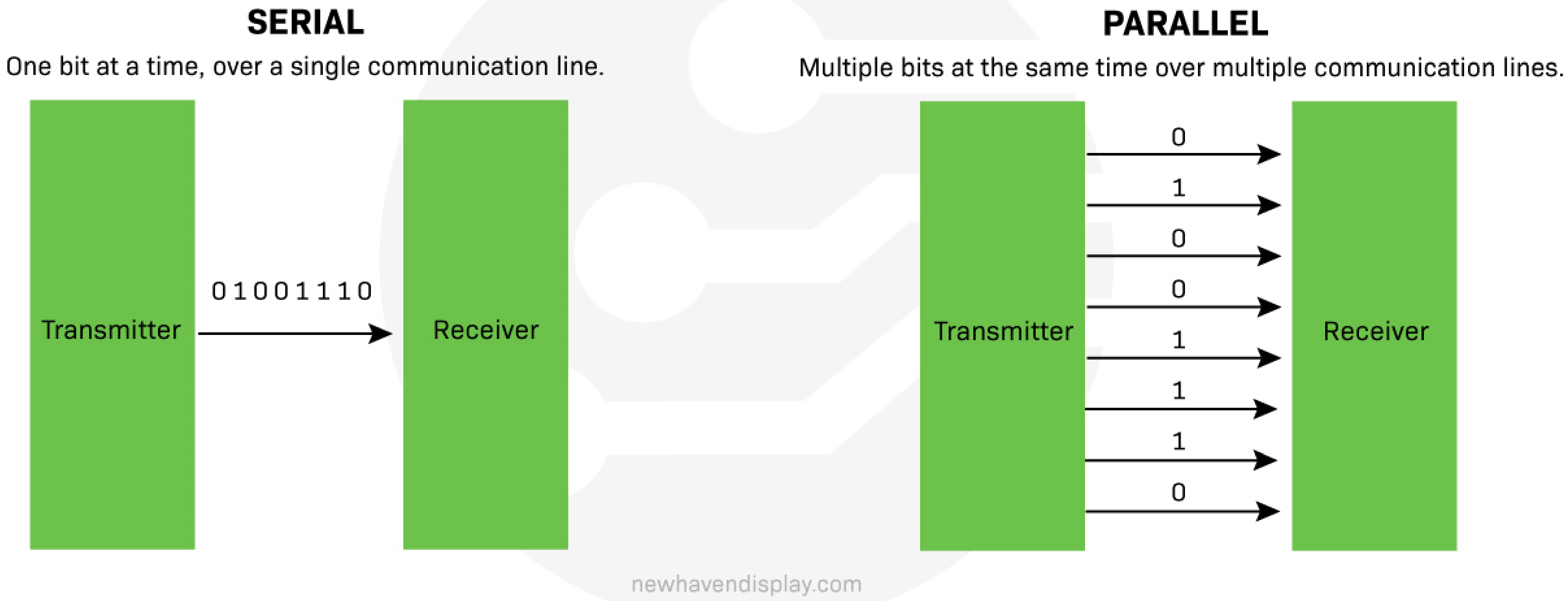
Serial communication

- Data bits are transmitted over the communication channel one after another (1 bit per CLK cycle) .
- Lower speed and frequency of operation
- Less number of cables needed (Hardware cost-effective)
- Convenient for long distances (a single wire) .
- Common Serial Protocols : **UART** , **I2C**, SPI , CAN , LIN .

Parallel communication

- Data bits are simultaneously transmitted over multiple communication channels (many bits per CLK cycle) .
- Faster speed and frequency of operation
- Higher number of cables/channels needed (more expensive hardware).
- Effective for short distances fast transfer.
- Common parallel Protocols : SA, ATA, SCSI, PCI and IEEE-488 .

Serial vs Parallel



Communication terminologies :

Bit Time : Bit time refers to the duration of time it takes to transmit a single bit of data in a digital communication system.

Bit Rate : number of bits transmitted per second. **Bit_rate = 1/Bit_time.**

Baud Rate : number of symbols (*group of multiple bits*) signaled/transmitted in a second. However, In some systems it is referenced as **bit rate (** *where each symbol is one bit only*). In serial communication systems, the baud rate should be set the same on both the transmitting and receiving devices to ensure proper data synchronization. **Symbol = defined number of bits Baud_rate = Bit_rate/ number_of_bits_per_symbol .**

Example: A device sends a 100 symbol (4-bits each) per second . The **Baud_rate** is 100 symbol/sec and **Bit_rate** is 400 bit/sec .

Outline :

- Introduction to communication
- **Serial communication**
- UART protocol
- UART example
- I2C protocol
- I2C Example

Serial communication Types

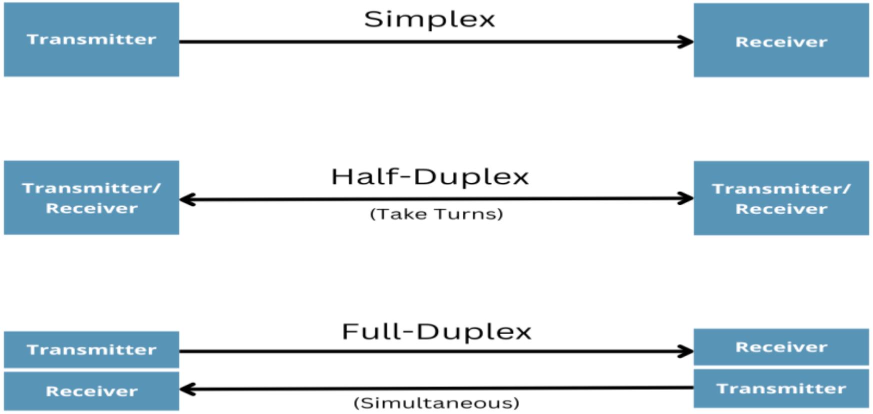
Serial communication is the most common across embedded systems for its low-cost and Hardware simplicity.
Serial communication is required to interface with :

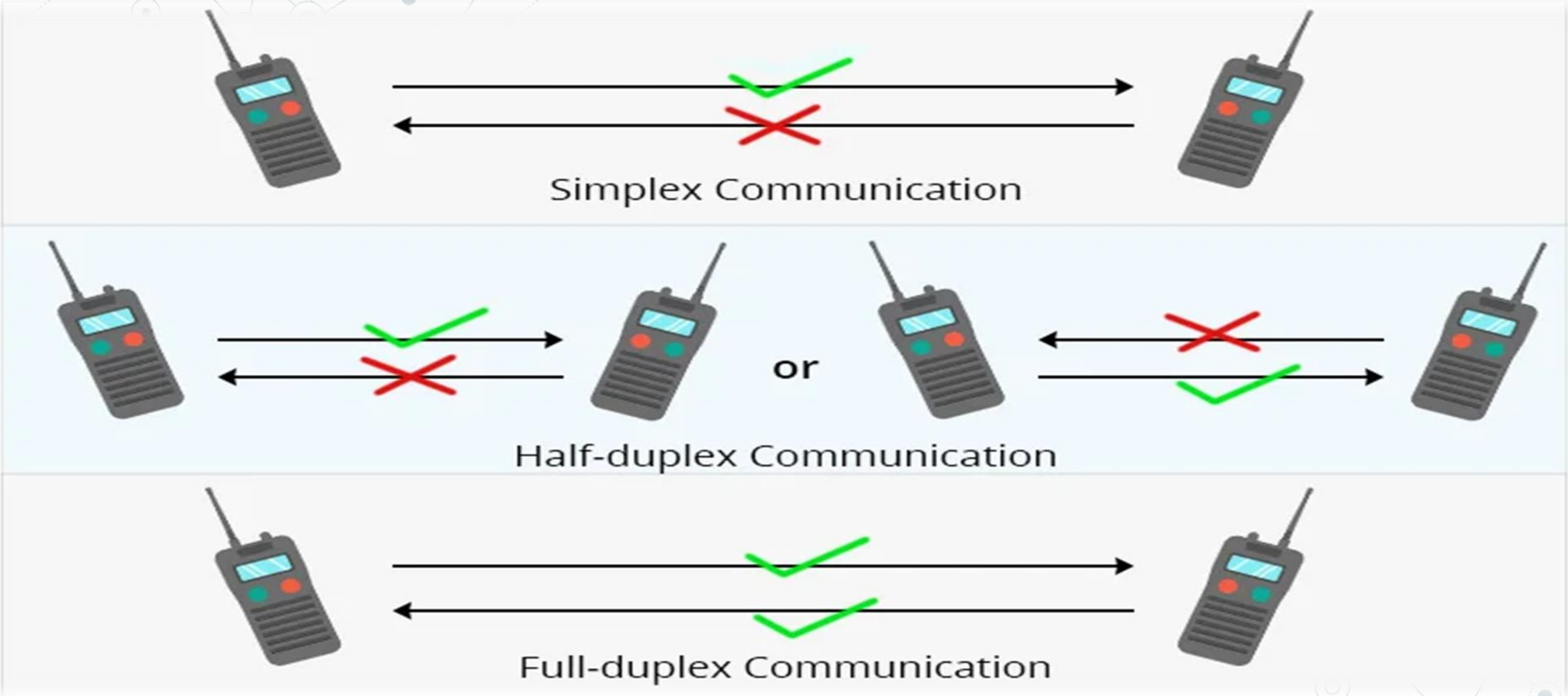
- Other microcontrollers and PCs .
- External Devices : LCD, GPS modules , External EEPROM, External ADC and many other peripherals.

✓ **Simplex Communication** : a One direction communication, the device will send data but cannot receive. (**TV, Radio**) .

✓ **Half-Duplex :** A device can both send and receive data but one at a time, it is a two-way direction communication but only one direction at a time.(**Walkie Talkie**)

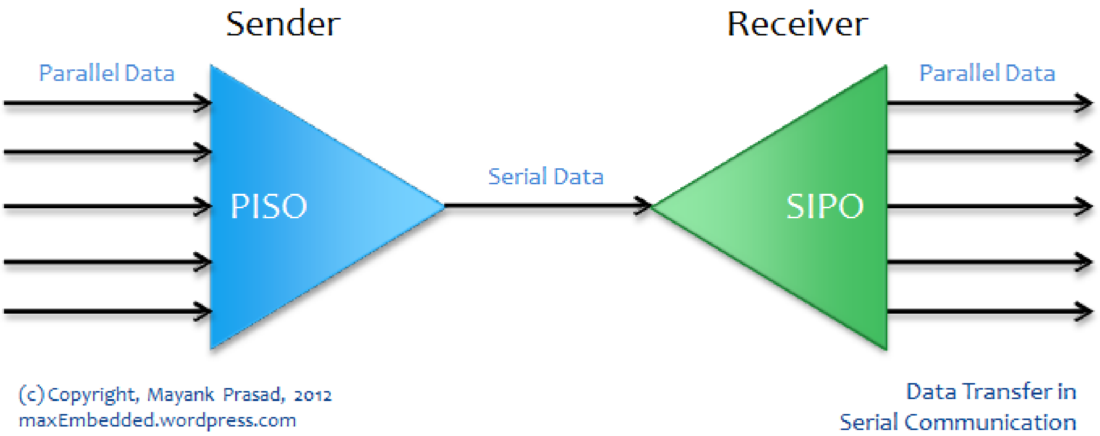
✓ **Full Duplex** : The devices can send/receive data simultaneously. (**mobile phone**) .




Simplex Communication
or
Half-duplex Communication
Full-duplex Communication

Data Transfer

Data transfer : Data existing at the microcontroller registers is in parallel form, as any bit can be accessed independent of its bit_ number , but in serial communication the Sender must serially transmit the bits one by one with the **MSB/LSB** bit first . This **PISO operation** (**Parallel In Serial Out**) can be done by shift-registers. However, the receiver device accepts the serial data and transform each bulk of successive bits into the ordinary parallel for to be stored and processed, this reverse operation is labelled as **SIPO** (**Serial In Parallel Out**) .



Serial communication

Synchronous

- In synchronous communication, the clock signal is used to ensure that both devices are operating at the same speed and to synchronize the timing of data transmission. (**SPI , I2C**)
- An extra CLK line is provided by the master device for synchronization and another data line carries the data.
- Less error frequency at short distances.
- No need for synchronization bits so only data bits are transferred so a faster baud rate is achieved.
- Clock skewing and additional hardware costs occur at long distances communication .

Asynchronous

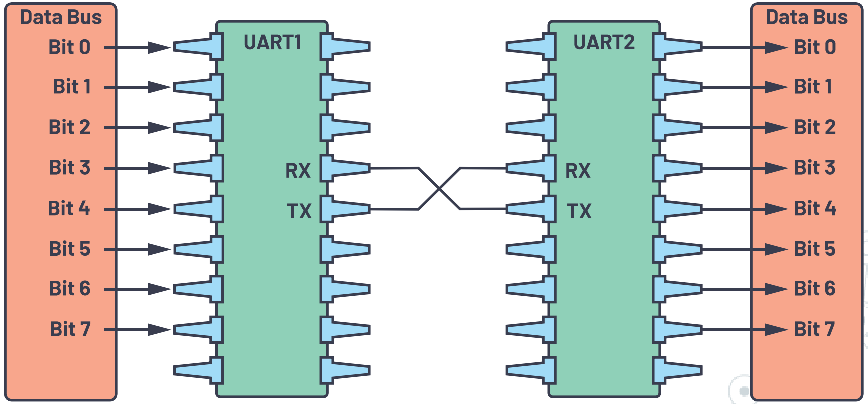
- Asynchronous communication does not rely on a shared clock signal for synchronization. Instead, it uses start and stop bits (synchronization bits) to indicate the beginning and end of each data frame **(UART**).
- No CLK line is needed. (less hardware)
- Synchronizations bits must be sent in each frame as each byte is encapsulated in such a frame. (lower bit rate).
- Devices should predefine the values of : **Baud_rate, Start_bits, Stop_bits, Error_checking_bits** .

Outline :

- Introduction to communication
- Serial communication
- **UART protocol**
- UART example
- I2C protocol
- I2C Example

UART

UART : UART (Universal Asynchronous Receiver-Transmitter) is a common hardware protocol used for asynchronous serial communication in embedded systems. It provides a straightforward method for transmitting and receiving data between devices using two communication lines: a transmit line (TX) for data transmission and a receive line (RX) for data reception.



<u>UART is Asynchronous Serial Full Duplex Protocol and this Induces the following :</u>

- Needs Only 2 I/O pins (Rx,Tx) for two data-lines .
- No shared CLK line
- Can receive and send data simultaneously.
- Must define a certain **Buad_rate** before starting the Communication.
- Must use synchronization bits in each data frame .
- USART (Universal Synchronous /Asynchronous Receiver-Transmitter) provides a synchronous mode of comm. where it uses an extra shared CLK line but it is not commonly used among the embedded applications.

UART Frame

In UART (Universal Asynchronous Receiver-Transmitter) communication, data is transmitted in frames. A frame consists of several components that together form a complete unit of data transmission. The components of a UART frame typically include:

- Start Bit: The start bit marks the beginning of a frame and is always a logic low (0) signal.
- **Data Bits**: The data bits carry the actual information being transmitted. The number of data bits in a frame can vary, Common configurations include 4,5,6,7,8,9 bits, but usually a **byte (8-bits)** is selected.
- Parity Bit (optional): The parity bit is an optional component used for error detection. It can be included in the frame to enable parity checking.
- Stop Bit(s): 1 or 2 bits to indicate the end of the frame and is always a logic high (1) .
- A 11-bit frame is shown below, as we can observe we need to send 11-bits in order to deliver a single byte of data

Start Bit '0'	**D0**	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**Parity Bit**	**Stop Bit '1'**

UART Frame

- When the Tx (transmitter line) is **IDLE** it keeps a sequence of stop bits (HIGH Logic).
- UART protocol allows us to configure the number of **data bits, Baud_rate_value, Parity bit setting, number_of_stop_bits** by writing to its assigned registers.
- **Parity bit** is one of the simplest method of error detection, if the parity-bit is enabled the sender (Tx) will calculate and add the parity bit to the data bits. After receiving the data the receiver (Rx) will calculate the parity bit and compare it the sent parity bit and if it mismatches a Frame error will be raised.
- The parity can be either an Even or an Odd parity checker, and it can be configured by manipulating the registers as well.
- **Even Parity** : Even parity bit is '0' when the number of '1' bits is even and it is '1' if the number is odd.
- **Odd Parity** : Odd parity bit is '0' when the number of '1' bits is odd and it is '1' if the number is even.
- As shown in the example below the number of '1' in data bits is 5 (odd) so the even parity-bit is 1 .
- If the Odd parity checker was configured , the parity bit would be '0' bit .

Start Bit '0'	D0 0	D1 1	D2 1	D3 0	D4 1	D5 1	D6 1	D7 0	Even/odd Parity Bit '1' / '0'	Stop Bit '1'

UART Frame

Advantages	disadvantages
Simple Hardware	Data frame is limited (1 byte)
Full duplex with only 2 wires	Low speed
Does not require a shared CLK	Does not support communication with multiple devices
Integrated Error checking	Devices must agree on a preset Baud_rate , Synchronization Bits

Start Bit ‘0’	D0 0	D1 1	D2 1	D3 0	D4 1	D5 1	D6 1	D7 0	Even/odd Parity Bit ‘1’ / ‘0’	Stop Bit ‘1’

UART Frame

Transmission steps :

1. Send a Start Bit (low).
2. Send Data bits (usually one byte with LSB first).
3. Calculate the parity Bit (optional).
4. Send Parity bit (optional)
5. Send Stop bit/s (High)

Reception steps :

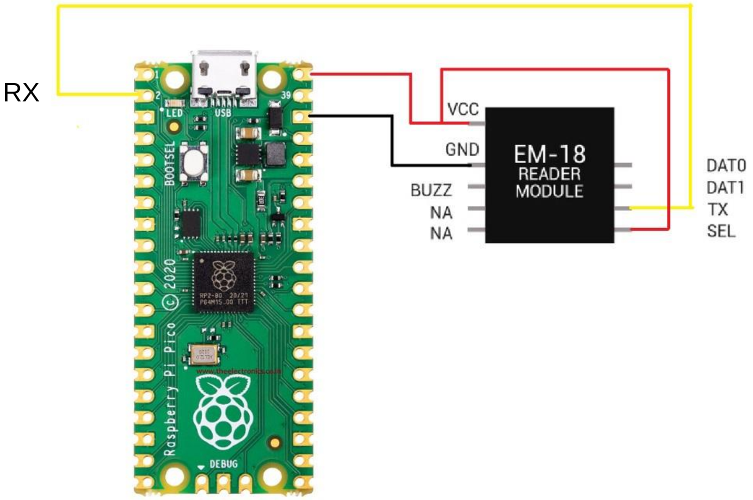
1. Receive Start bit
2. Receive Data bits
3. receive Parity bit (optional)
4. Calculate the parity Bit (optional).
5. Check the parity-bit match/mismatch
6. receive Stop bit/s
7. Discard start, stop and parity bits to have only pure data byte .

Start Bit '0'	D0 0	D1 1	D2 1	D3 0	D4 1	D5 1	D6 1	D7 0	Even/odd Parity Bit '1' / '0'	Stop Bit '1'

Outline :

- Introduction to communication
- Serial communication
- UART protocol
- **UART example**
- I2C protocol
- I2C Example

```c
#include "hardware/uart.h"

#define UART_ID uart0
#define UART_BAUD_RATE 9600
#define UART_RX_PIN 1
#define UART_TX_PIN 0

int main() {
    stdio_init_all();

    // Initialize UART
    uart_init(UART_ID, UART_BAUD_RATE);
    gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);
    gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
    uart_set_hw_flow(UART_ID, false);
    uart_set_format(UART_ID, 8, 1, UART_PARITY_NONE);

    while (1) {
        // Read data from RFID sensor
        char data = uart_getc(UART_ID);

        // Process and utilize the RFID data
        printf("Received RFID data: %c\n", data);

        // Other processing or actions based on the RFID data

        sleep_ms(100);  // Delay between reads
    }

    return 0;
```



UART communication between a raspberry-pi pico and RFID sensor where the pico receives the sensory data through the Rx pin using UART protocol.

Communication between 2 microcontrollers (Pico) using UART

Sender Code

```c
#define UART_ID uart0
#define BAUD_RATE 9600
#define UART_TX_PIN 0 // Choose a GPIO pin for TX

int main() {
    stdio_init_all();

    // Initialize UART with specified baud rate
    uart_init(UART_ID, BAUD_RATE);

    // Set UART TX pin
    gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
    uart_set_hw_flow(UART_ID, false, false); // Disable hardware flow control

    // Set UART configuration (default is 8N1)
    uart_set_format(UART_ID, 8, 1, UART_PARITY_NONE);

    while (1) {
        uart_puts(UART_ID, "Hello from Transmitter!\n");
        sleep_ms(1000); // Delay between transmissions
    }

    return 0;
}
```

Receiver code

```c
#define UART_ID uart0
#define BAUD_RATE 9600
#define UART_RX_PIN 1 // Choose a GPIO pin for RX

int main() {
    stdio_init_all();

    // Initialize UART with specified baud rate
    uart_init(UART_ID, BAUD_RATE);

    // Set UART RX pin
    gpio_set_function(UART_RX_PIN, GPIO_FUNC_UART);

    // Set UART configuration (default is 8N1)
    uart_set_format(UART_ID, 8, 1, UART_PARITY_NONE);

    while (1) {
        while (uart_is_readable(UART_ID)) {
            printf("Received: %c\n", uart_getc(UART_ID));
        }
    }

    return 0;
}
```

Outline :

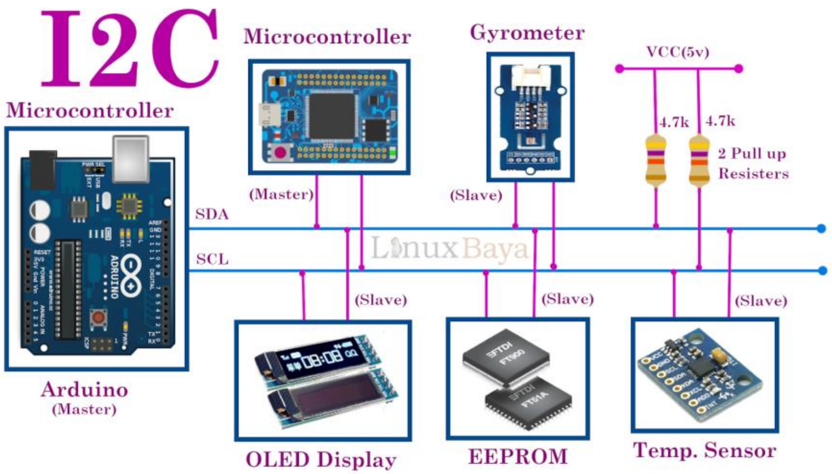
- Introduction to communication
- Serial communication
- UART protocol
- UART example
- **I2C protocol**
- I2C Example

I2C Protocol

I2C : The Inter-Integrated Circuit (I2C) protocol is a widely used serial communication protocol that allows multiple devices to communicate with each other using just two wires: a data line (SDA) and a clock line (SCL). In I2C, devices are either masters or slaves. The master initiates communication and controls the bus, while slaves respond to commands from the master.

<u>I2C/TWI is synchronous Serial half Duplex Protocol and this Induces the following :</u>

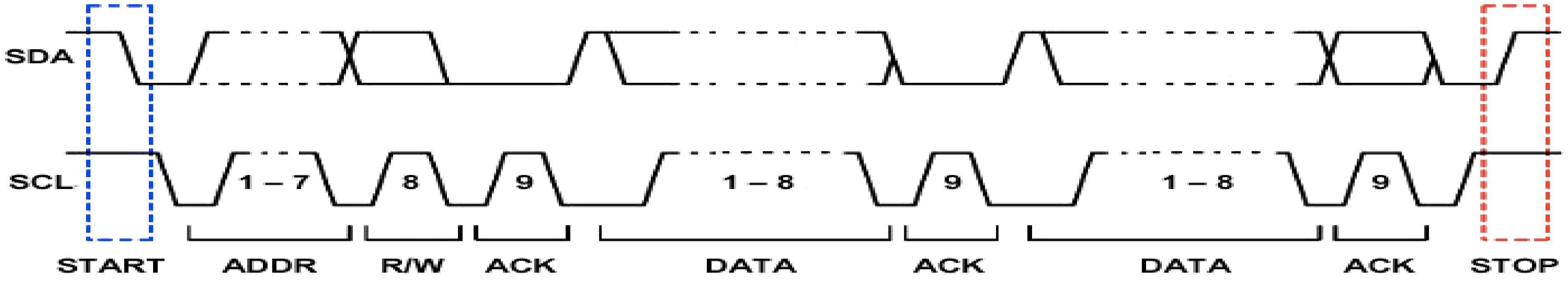
- Needs Only two wires SCL (Serial clock line) and SDA (Serial Data Line) .
- Half duplex states that the data flow is either form Master to slave or from Slave to Master.
- It is a multiple master multiple slave protocol indicating That multiple devices can communicate .
- No synchronization bits needed.
- No Predefined baud_rates as the synchronization is achieved using the CLK signals and acknowledgements.



I2C Protocol

Handshaking protocol:

- Acknowledge (ACK): After the master sends either the device address or data bytes to the slave, the receiving device (either the slave or the master, depending on the direction of communication) sends an acknowledgment bit. If the receiving device acknowledges by pulling the SDA line low (‘0’) during the **ACK** slot, it indicates it's ready to proceed with further data transfer. **ACK = 0**
- Conversely, if the receiving device doesn't acknowledge by keeping the SDA line high, the **No Acknowledge (NACK) (HIGH)** indicates the end of communication (either the end of a transmission or that the slave device didn't recognize the address or data sent). **NACK = 1**
- **Each byte is followed by either an ACK or a NACK .**



I2C Protocol

I2C starting condition and initialization steps :

- Only a single master can be active at a time.
- SDA and SCL buses are bidirectional as master can write to slave and the slave can write to it but at different instances .
- All devices are given unique addresses of 7-bits (up to 128 device) .

I. Initially SDA and SCL are Pulled to High Logic (open-drain system) .

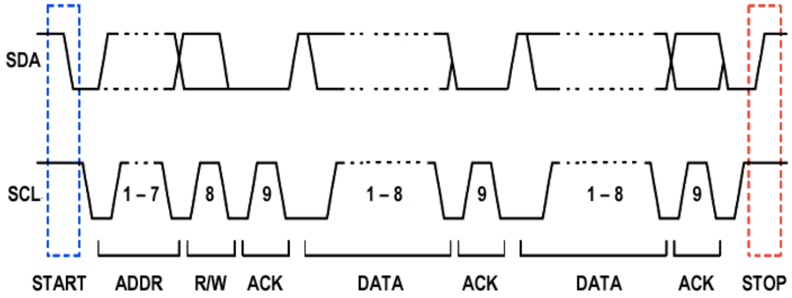
II. In order for a device to be a Master it must claim the bus by first pulling The SDA line Low while the SCL is HIGH and consequently pulling the SCL low.

III. All slaves become active when a master Claims the bus and

IV. The master send a 7-bit address of the Targeted slave

V. The address is followed by a read/write bit
Read --- 1
Write --- 0

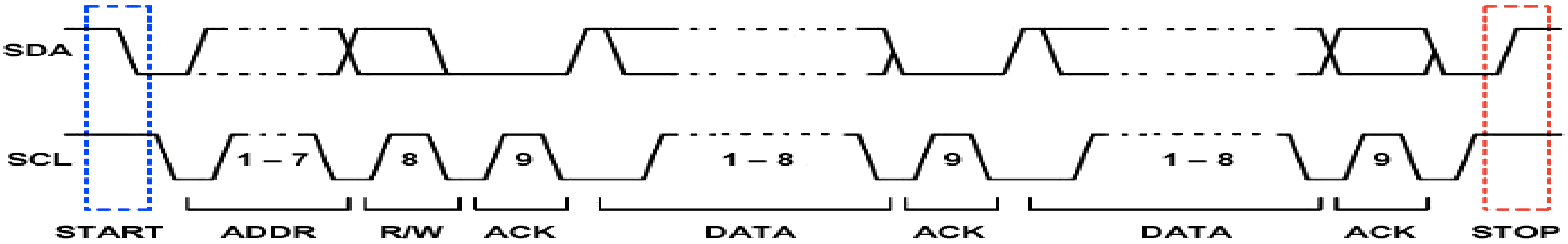
VI. All the slaves compare their addresses with the sent address. and only the targeted slaves replies with an ACK (‘0’) Low Bit .
The master receives a NACK if the address is wrong or
A corrupted communication occurs.



I2C Protocol

I2C stopping condition :

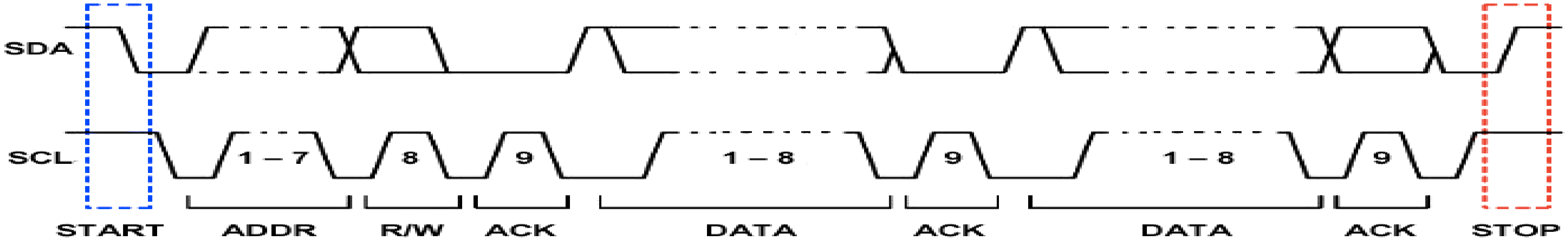
- After the first ACK the master sends the data in 8-bit blocks (bytes) with **MSB** first and waits for an ACK by the slave till the whole data is transmitted or an error occurs so the NACK is received. (data transmission is done only when the SCL is LOW)
- The communication is stopped when the master generates the stop condition by pulling the SDA HIGH when the SCL is HIGH and consequently pulling the SCL low .
- After the master does the stop conditions, any device can initiate the starting conditions to claim the bus.



I2C Protocol

I2C Protocol sequence

1. Master initiates the CLK
2. Master send the Starting condition (SDA = LOW , START = ‘0’) before the SCL goes LOW. (arbitration)
3. Master sends the 7-bit unique fixed address of the targeted slave
4. The Master sends the R/W bit . (0 = master writes , 1 = slave writes to master (master read from slave))
5. An Ack is sent by the slave . ACK = 0
6. Then the transmitter (master/slave dependent on the R/W bit) sends the data block (1 byte of data) with the MSB first
7. followed by an ACK from the receiver
8. This pattern (6,7) is repeated till the data is transferred .
9. The master pulls the SDA HIGH when the SCL is HIGH followed by the SCL pulled to LOW to STOP the condition .

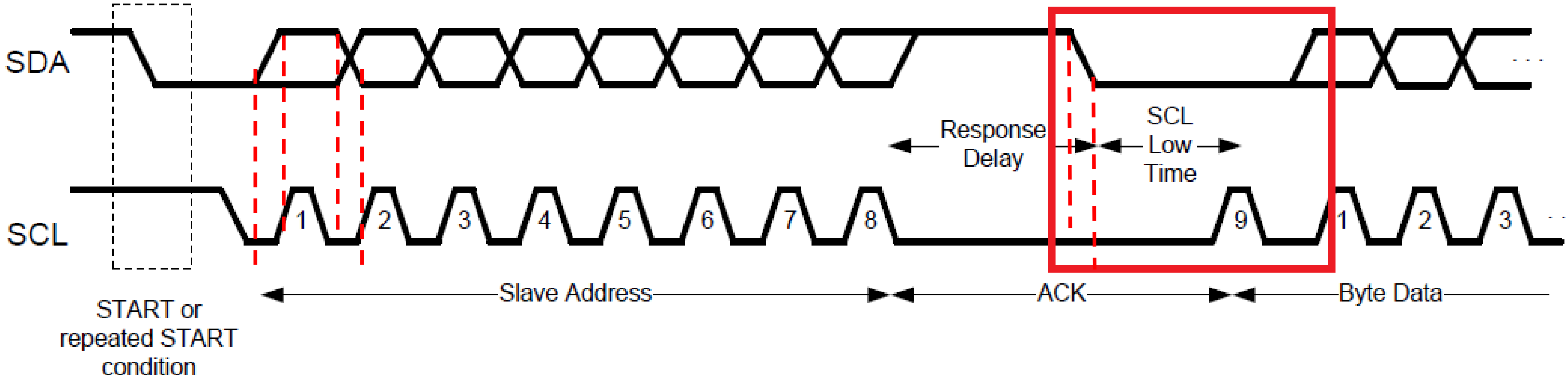


Special cases

a) Clock stretching is an essential feature in the I2C (Inter-Integrated Circuit) protocol that allows slave devices to slow down communication by holding the clock line (SCL) low. This feature is particularly useful when a slave device needs more time to process data (either to store the byte or to send the ACK) before responding to the master.

SDA value (bit value) can only change when the **CLK is LOW** (waits for the High- low edge)

The slave is not ready for more data so it buys time by holding the SCL low so that the master will wait for the CLK High edge to come to process the ACK so that it can proceed

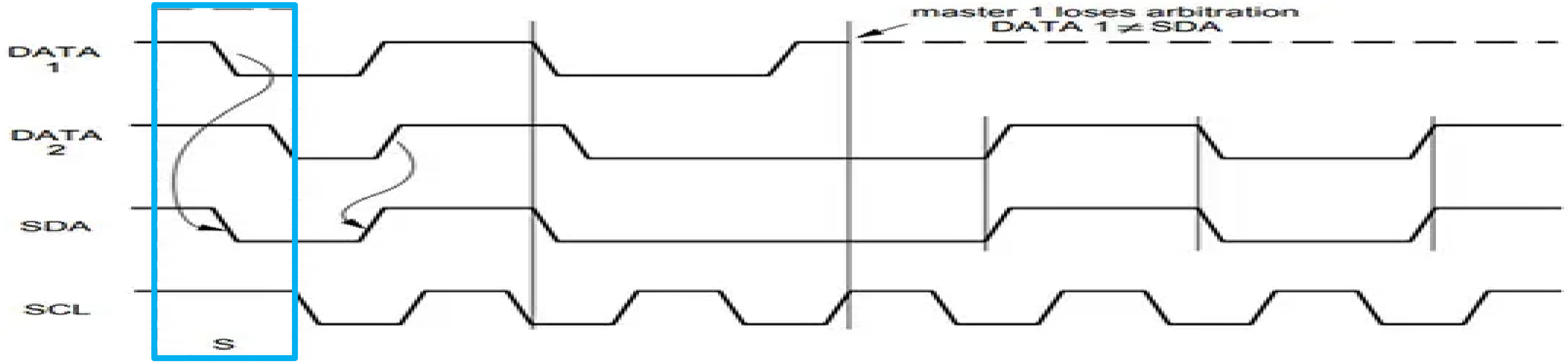


Special cases

b) I2C arbitration: is a mechanism within the I2C protocol that resolves conflicts when multiple masters attempt to access the bus simultaneously. Since I2C supports multiple masters, arbitration ensures orderly access to the bus, preventing data corruption and bus contention.

- In a multi-master I2C system, more than one master device can try to access the bus simultaneously. Each master sends its data or address onto the bus and concurrently monitors the bus for any collisions. Masters continuously monitor the SDA (data line) and SCL (clock line) to detect conflicts. It occurs when two or more devices do the start condition in the same time.

Both masters are worthy of the bus as both did the starting condition

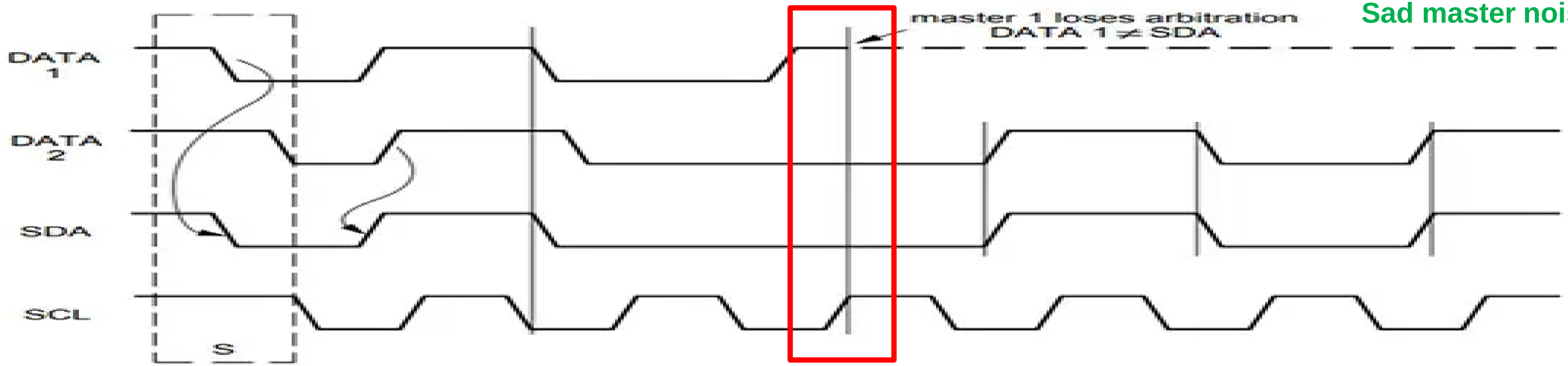


Special cases

Collision : If a master sends a '1' on the bus and senses a '0', indicating that another master has sent a '0', a **collision** is detected

Arbitration Decision: When a collision occurs, masters compare the bits they transmitted with the bits they read from the bus. The master transmitting the bit that was not read back (its '1' was overridden by another master's '0') realizes it has lost arbitration. (**The winning master is the one with first Low bit transmitted**).

Recovery and Retransmission: The master that lost arbitration releases the bus, allowing the winning master to continue transmission. The losing master waits for the bus to become idle and then retries its transmission.



Sad master noises ☹

I2C

Advantages	disadvantages
Simple Hardware Requirements	Complex protocol
Multi-Master Capability	Low speed
Synchronous Communication so less error frequency	Limited distance die to high cost for the open drain system
Clock Stretching	Clock skewing occur at long distances
Up to 127 addressable devices	In a multi-master scenario, simultaneous access attempts can lead to contention
Low Pin Count only 2 pins are needed and devices don’t agree on a preset parameters	Increase power dissipation as it includes pull-up resistors

Outline :

- Introduction to communication
- Serial communication
- UART protocol
- UART example
- I2C protocol
- **I2C Example**

Write a C code using the Raspberry Pi Pico's SDK to read 5 bytes from an EEPROM and then write 5 bytes to the same EEPROM using I2C communication:

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#define I2C_PORT i2c0
#define EEPROM_ADDR 0x50 // Replace with your EEPROM's address
#define READ_LENGTH 5
int main() {
    stdio_init_all();
    i2c_init(I2C_PORT, 100000); // Initialize I2C at 100 kHz
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    uint8_t read_data[READ_LENGTH];
    uint8_t data_to_write[READ_LENGTH] = {0x11, 0x22, 0x33, 0x44, 0x55}; // Data to write

    // Read from EEPROM
    i2c_write_blocking(I2C_PORT, EEPROM_ADDR, "\x00", 1, true); // Set EEPROM address pointer to 0x00
    i2c_read_blocking(I2C_PORT, EEPROM_ADDR, read_data, READ_LENGTH, false); // Read 5 bytes from EEPROM

    printf("Read from EEPROM: ");
    for (int i = 0; i < READ_LENGTH; ++i) {
        printf("%02x ", read_data[i]);
    }
    printf("\n");
    // Write to EEPROM
    i2c_write_blocking(I2C_PORT, EEPROM_ADDR, "\x00", 1, true); // Set EEPROM address pointer to 0x00
    i2c_write_blocking(I2C_PORT, EEPROM_ADDR, data_to_write, READ_LENGTH, false); // Write 5 bytes to EEPROM
    return 0;
}
```

I have to write first to select the memory address that I can read from.

Communication between 2 microcontrollers (Pico) using I2C

The master sends the "Hello" message to the slave, the slave receives it and prints it, and then the slave sends the "World" message back to the master, which the master then receives and prints.

Master Code

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

#define I2C_PORT i2c0
#define I2C_ADDR 0x12
#define BUFFER_SIZE 8

int main() {
    stdio_init_all();
    i2c_init(I2C_PORT, 100000);
    gpio_set_function(4, GPIO_FUNC_I2C);
    gpio_set_function(5, GPIO_FUNC_I2C);
    gpio_pull_up(4);
    gpio_pull_up(5);

    uint8_t send_buffer[BUFFER_SIZE] = "Hello";
    uint8_t receive_buffer[BUFFER_SIZE];

    while (1) {
        i2c_write_blocking(I2C_PORT, I2C_ADDR, send_buffer, BUFFER_
SIZE, false);
        printf("Sent: %s\n", send_buffer);

        sleep_ms(1000);

        i2c_read_blocking(I2C_PORT, I2C_ADDR, receive_buffer, BUFFE
R_SIZE, false);
        printf("Received: %s\n", receive_buffer);
    }
}
```

Slave code

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

#define I2C_PORT i2c1
#define I2C_ADDR 0x12
#define BUFFER_SIZE 8

int main() {
    stdio_init_all();
    i2c_init(I2C_PORT, 100000);
    gpio_set_function(2, GPIO_FUNC_I2C);
    gpio_set_function(3, GPIO_FUNC_I2C);
    gpio_pull_up(2);
    gpio_pull_up(3);

    uint8_t receive_buffer[BUFFER_SIZE];
    uint8_t send_buffer[BUFFER_SIZE] = "World";

    i2c_set_slave_mode(I2C_PORT, true, I2C_ADDR);

    while (1) {
        uint len = i2c_read_blocking(I2C_PORT, I2C_ADDR, receive_bu
ffer, BUFFER_SIZE, false);
        printf("Received: %.*s\n", len, receive_buffer);

        sleep_ms(1000);

        i2c_write_blocking(I2C_PORT, I2C_ADDR, send_buffer, BUFFER_
SIZE, false);
        printf("Sent: %s\n", send_buffer);
    }
}
```

THANK YOU