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

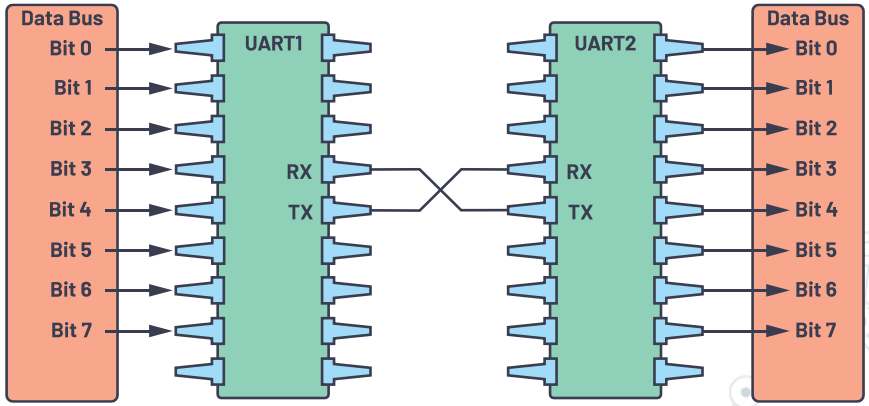
Office: C7.201
Mail : Maysarah.mohamed@guc.edu.eg

Outline :

- **Recap**
- Interrupts
- External vs Internal
- Interrupts handling
- Examples

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.

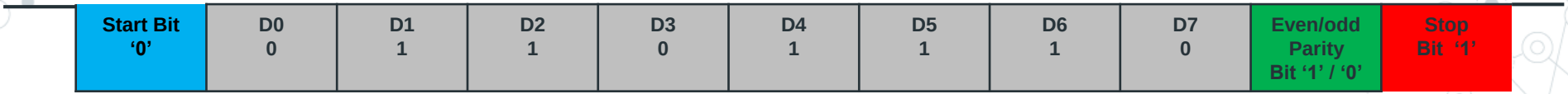
UART Frame

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

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 .



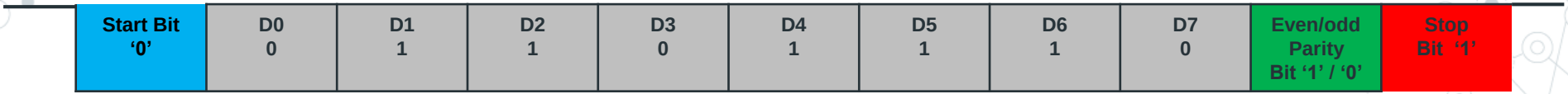
UART Frame

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 .

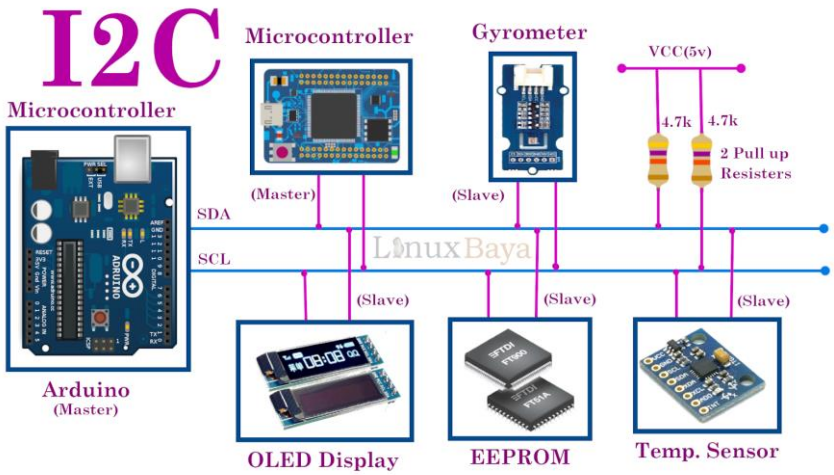


I2C Protocol

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.

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

- 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

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) .

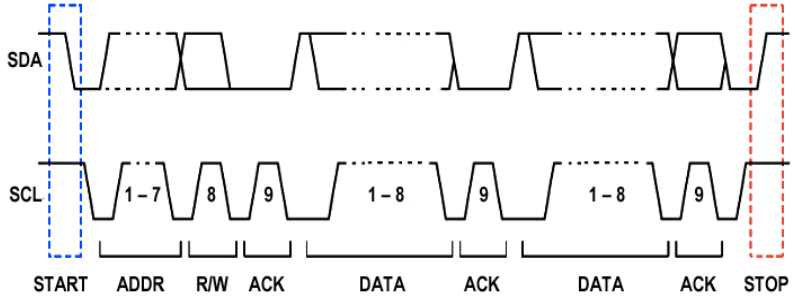
II. In order for a device to be a Master it must claim the bus by first pulling The SDA line Low while the SCL is HIGH and consequently pulling the SCL low.

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

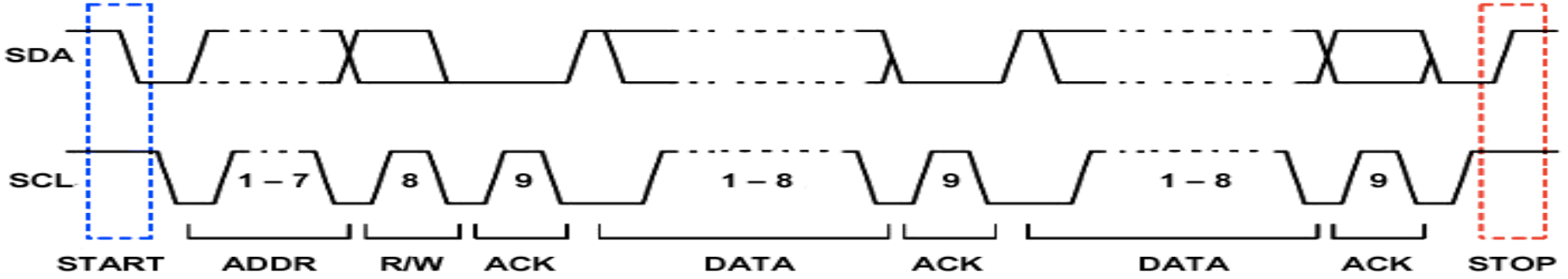
VI. All the slaves compare their addresses with the sent address. and only the targeted slaves replies with an ACK (‘0’) Low Bit . The master receives a NACK if the address is wrong or A corrupted communication occurs.



I2C Protocol

Handshaking protocol:

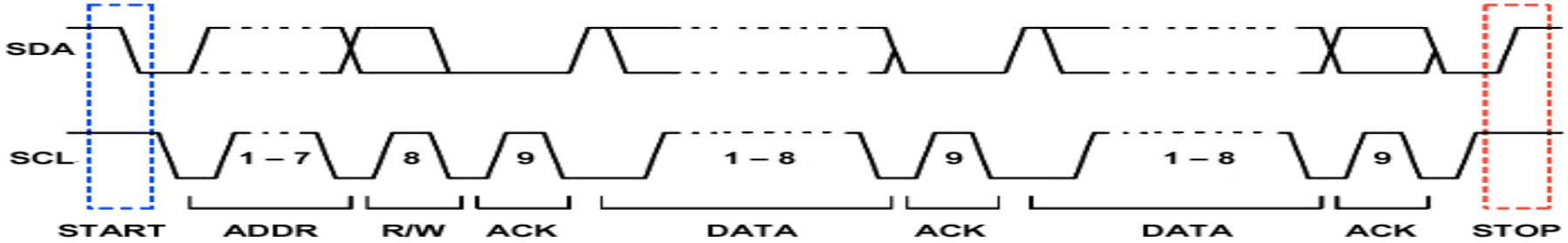
- Acknowledge (ACK): After the master sends either the device address or data bytes to the slave, the receiving device (either the slave or the master, depending on the direction of communication) sends an acknowledgment bit. If the receiving device acknowledges by pulling the SDA line low (‘0’) during 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 stopping condition :

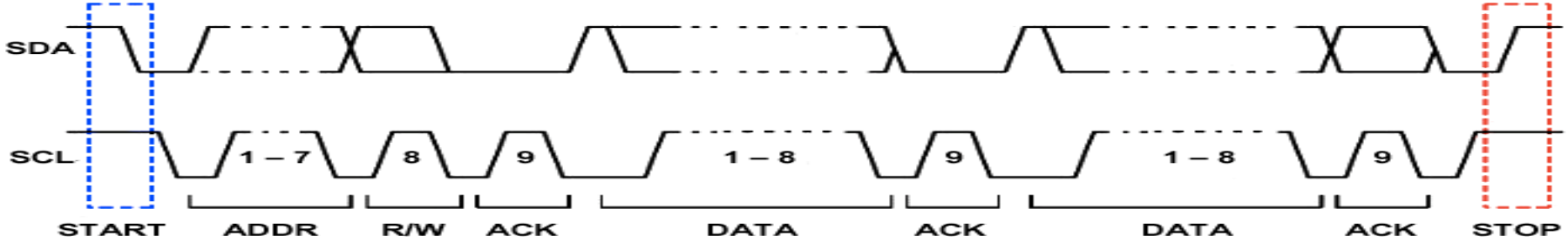
- After the first ACK the master sends the data in 8-bit blocks (bytes) with **MSB** first and waits for an ACK by the slave till the whole data is transmitted or an error occurs so the NACK is received. (data transmission is done only when the SCL is LOW)
- The communication is stopped when the master 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 .

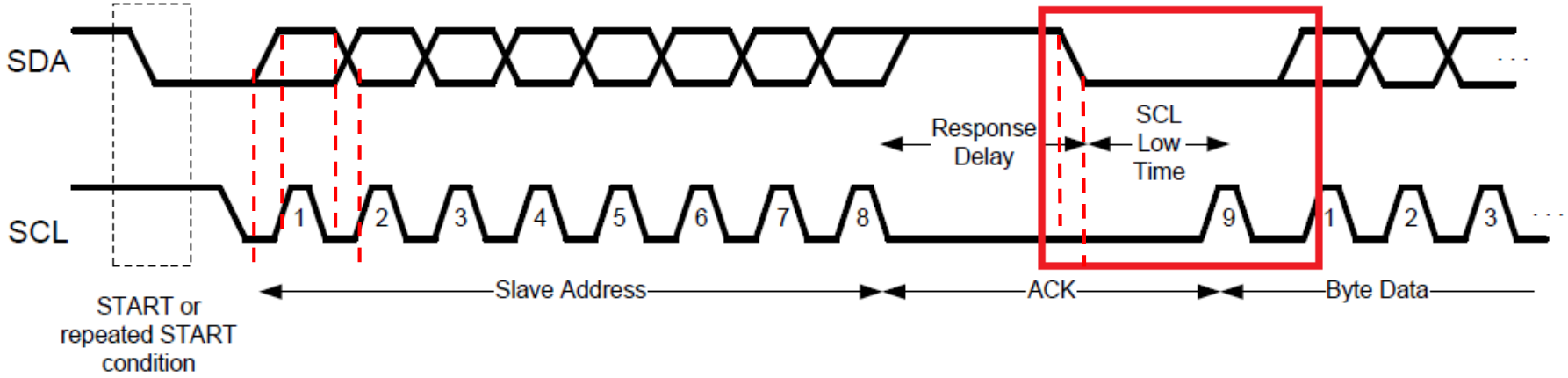


Special cases

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

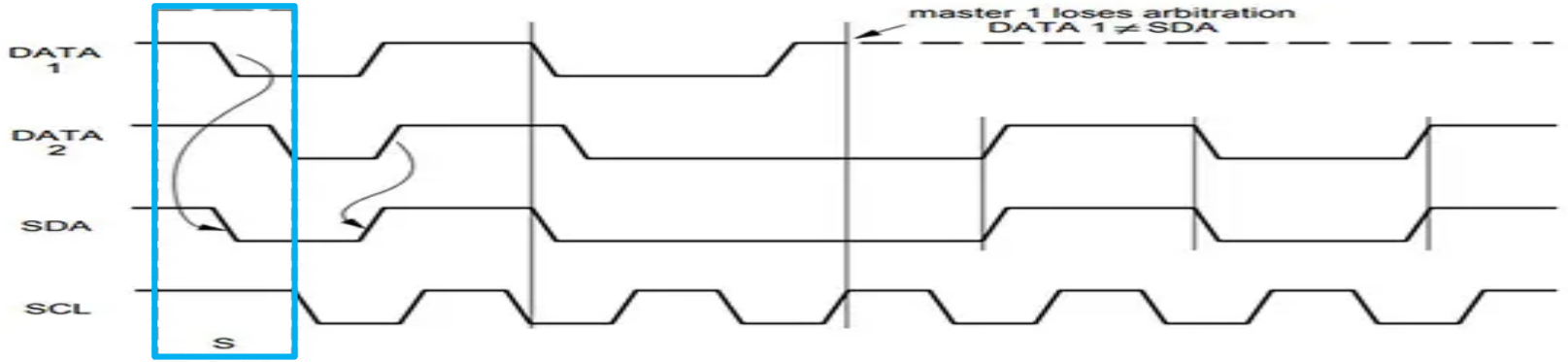


Special cases

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

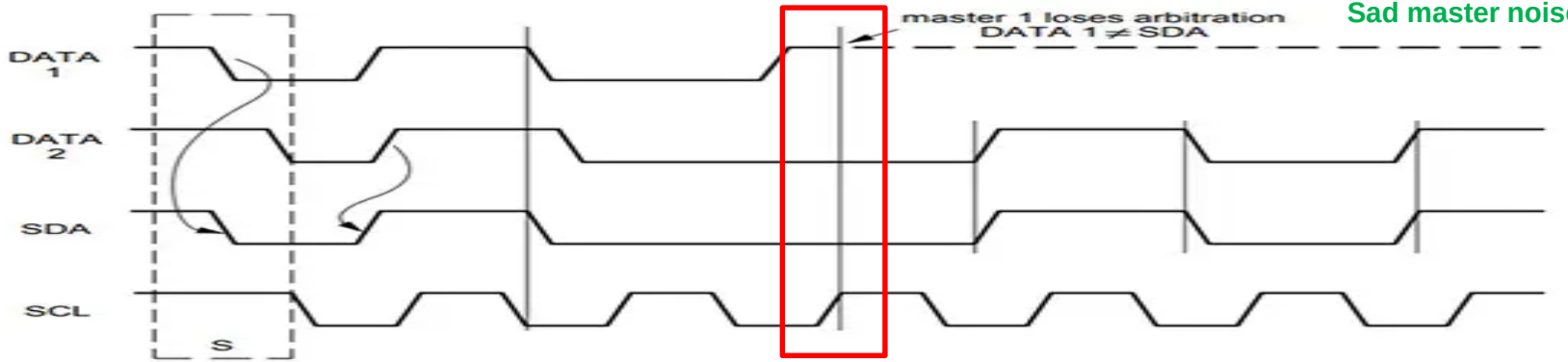


Special cases

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.



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

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

```c
#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

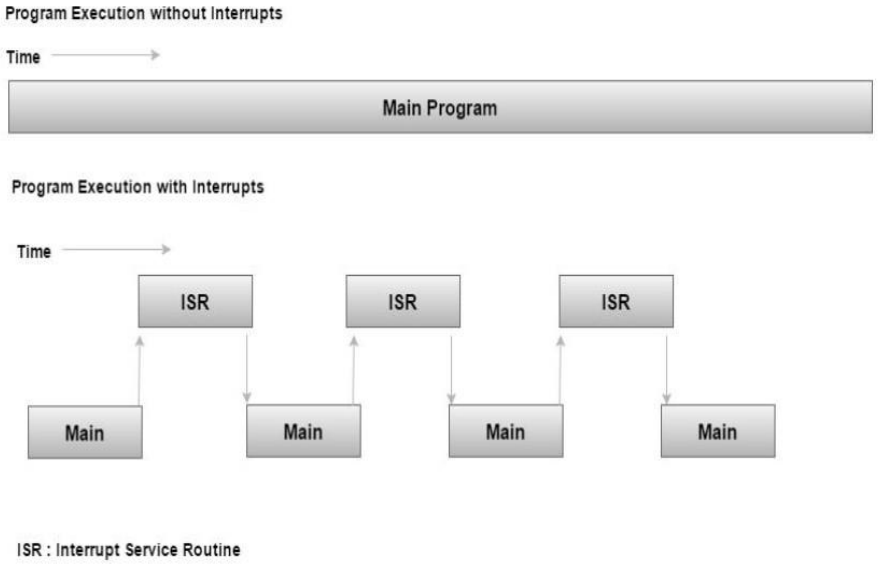
```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

#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);
    }
}
```

Outline :

- Recap
- **Interrupts**
- External Vs internal
- Interrupt Examples

What are Interrupts?

- Interrupts are basically external/internal **events** that require **immediate** attention by the microcontroller.
- It is basically a signal generated by a software (internal interrupt) or by a hardware (external interrupt).
- When an interrupt occurs, the microcontroller pauses any current tasks and immediately switch its context to handle the interrupt by executing interrupt-related tasks, such tasks are labelled as **Interrupt Service Routine (ISR).**
- After finishing the **ISR,** the microcontroller switches back to its normal tasks.



Internal Interrupts

Internal interrupts, also known as software interrupts or exceptions, are events that occur within the microcontroller's core itself, triggered by specific conditions or instructions during program execution. These interrupts are typically generated by the processor or specific internal components and are used for various purposes.

Common Types of Internal Interrupts:

System Timer Interrupt (SysTick): Generated by the system timer peripheral in the microcontroller. Used for system timing, scheduling tasks, or operating system tick handlers.

Memory Management Fault: Triggered when there's an error related to memory access or management (e.g., invalid memory access, permissions violation).

Hard Fault: Indicates critical faults or errors, such as accessing invalid memory addresses or division by zero.

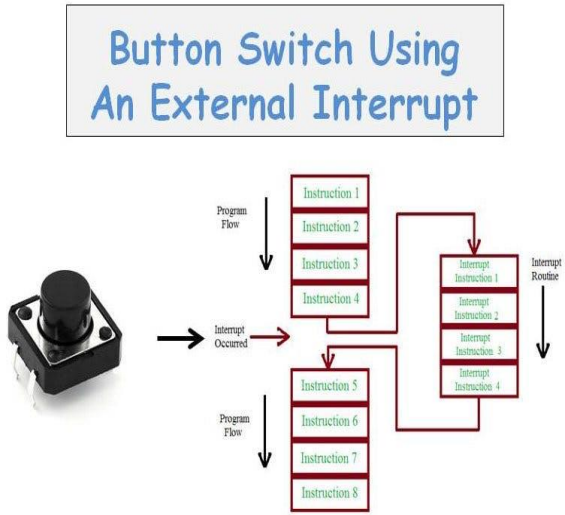
Interrupted Exceptions: Special instructions or events that interrupt normal execution (e.g., divide-by-zero exception, undefined instruction exception).

External Interrupts

External interrupts can be an input event to our GPIO pins from a sensor, Keypad or a pushbutton, when triggered it asserts an interrupt request to the processor to suspend its current task and do The ISR piece of code. The event can be (Rising_Edge , Falling_edge, Change).

Common Types of External Interrupts:

- **GPIO Interrupts:** Triggered by changes in the state of GPIO pins (e.g., rising edge, falling edge, or both).
- **Timer Interrupts :** Generated by timers or counters when they reach a specific value or overflow. Used for time-sensitive operations, periodic tasks, or event scheduling.
- **Communication Interface Interrupts:** For serial communication (UART, SPI, I2C, etc.): Generated when data is received, transmitted, or when specific conditions are met (e.g., buffer full/empty).Often used for communication with external devices.
- **Peripheral Interrupts:** Many microcontrollers have built-in peripherals (e.g., ADC, PWM, DMA) that can generate interrupts upon completion of operations, errors, or specific events.
- **External Interrupt Controller (EIC)** Inputs: Some microcontrollers have dedicated external interrupt controllers that allow external devices to directly trigger interrupts.



Outline :

- Recap
- Interrupts
- External vs Internal
- **Interrupts handling**
- Examples

Interrupts Handling

External interrupts can be an input event to our GPIO pins from a sensor, Keypad or a pushbutton, when triggered it asserts an interrupt request to the processor to suspend its current task and do The ISR piece of code. The event can be (Rising_Edge , Falling_edge, Change).

Handling External Interrupts:

Interrupt Service Routines (ISRs): Each type of external interrupt usually has a corresponding ISR that handles the interrupt when it occurs. (**all interrupts are stored in ROM in Interrupt Vector Table**).
Interrupt Priorities: External interrupts can have different priority levels. Configuring priorities determines which interrupt gets serviced first when multiple interrupts occur simultaneously.
Interrupt Masking: Disabling or enabling interrupts selectively to control their handling during critical sections of code execution.

```
Int main() {
    disable_interrupts( ) ;  // disable the interrupts so critical section will not suspended
     // do all critical tasks
      x = x+5 ; ….etc       // critical section
    enable_interrupts() ;  } // enable interrupts for normal operating mode
```

Interrupts Handling in PICO Rp2040

- **ISR should be deterministic, short and executable in small time.**

- **All ISRs/handlers are void functions and are called automatically by the processor hardware if they are enabled when the interrupt event occurs.**

- **When the interrupts occurs the microcontroller does the following steps :**

1. It completes the execution of the current instruction.
2. Stores the address of the next instruction (the program counter register contents).
3. Microcontroller CPU jumps to the interrupt vector table to get the ISR address of the triggered interrupt (Context Switching)
4. The ISR/Handler function address is load in the program counter register
5. The CPU starts executing the ISR till it ends then the ISR flag gets resetted
6. The CPU switches back to the stored instruction address

Interrupts Handling in PICO Rp2040

All GPIO pins of Pico Rp2040 can be configured as external Interrupts

The RP2040 microcontroller used in the Raspberry Pi Pico has 32 interrupt vectors (5-bits) available for various peripherals and system events. These interrupt vectors are allocated to handle interrupts from different sources, including timers, GPIO interrupts, UART, I2C, SPI, DMA, and other peripherals integrated into the RP2040.

These 32 interrupt vectors can handle interrupts generated by different sources or events in the system. The interrupt controller within the RP2040 manages these vectors, allowing the CPU to respond to and handle various asynchronous events efficiently.

Each interrupt vector corresponds to a specific interrupt source or peripheral. By utilizing the interrupt controller and appropriately configuring the **interrupt priorities and handlers**, programmers can design systems that respond to interrupts from different peripherals or events in real-time or based on the priority of the events.

IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

Interrupts Handling in PICO Rp2040

In the RP2040 microcontroller used in the Raspberry Pi Pico, interrupt priorities can be configured for different interrupt sources. The RP2040 supports up to 32 priority levels for interrupts, allowing you to set the priority of each interrupt source relative to others

Handling Interrupt Priorities:RP2040 allows setting priorities from 0 (highest) to 31 (lowest) for each interrupt source. Lower numerical values represent higher priority. Interrupts with lower priority numbers have higher priority over those with higher numbers.

Interrupt Controller (IRQ):
The RP2040 uses an Interrupt Controller (IRQ) to manage interrupt priorities. Each interrupt source has its own interrupt priority level.

Exception	Vector	Core	Priority
1	Reset	M0/M4/M3/M7	-3
2	NMI	M0/M4/M3/M7	-2
3	HardFault	M0/M3/M4/M7	-1
4	MemManageFault	M3/M4/M7	User
5	BusFault	M3/M4/M7	User
6	UsageFault	M3/M4/M7	User
7-A	Reserved		
B	SVCall	M0/M3/M4/M7	User
C	Debug Monitor	M3/M4/M7	User
D	Reserved		
E	PendSV	M0/MM3/M4/M7	User
F	SysTick	M0(optional)/M3/M4/M7	User
10-...	IRQ0, IRQ1, ... (Vendor)	M0: 0x10-0x47 M4/M7: IRQ0-239	User

Outline :

- Recap
- Interrupts
- External vs Internal
- Interrupts handling
- **Examples**

Critical Section handling

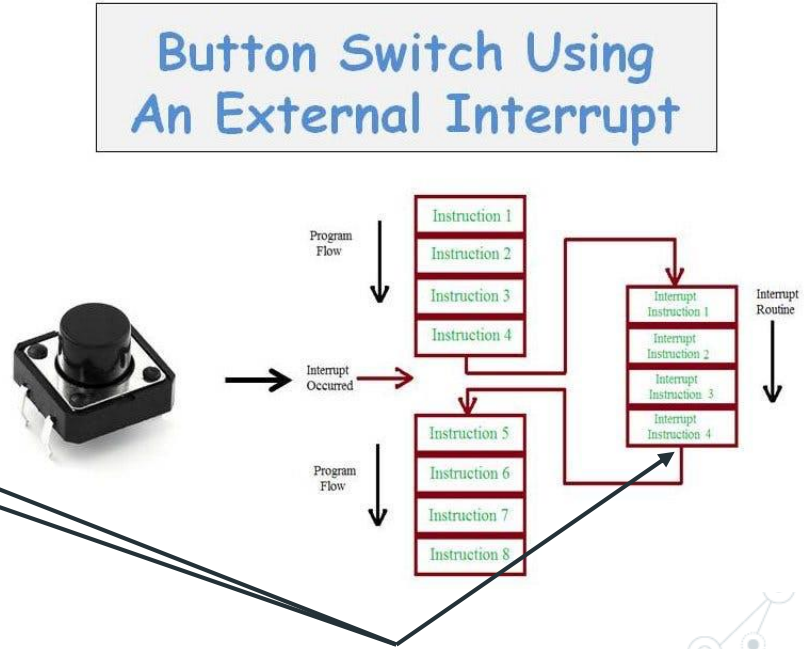
```c
#include <stdio.h>
#include <stdint.h>
#include "hardware/irq.h"

volatile uint32_t criticalVariable = 0;  // Shared variable

void criticalSection() {
    // Disable interrupts to protect critical section
    irq_set_enabled(false);

    // Critical operations go here
    criticalVariable++;  // Incrementing a shared variable

    // Re-enable interrupts after critical section
    irq_set_enabled(true);
}

int main() {
    // Simulating some operations in a loop
    for (int i = 0; i < 10; ++i) {
        criticalSection();
        printf("Critical variable value: %d\n", criticalVariable);
    }

    return 0;
}
```

irq_set_enabled(true) // enable interrupts

irq_set_enabled(true) // disable interrupts

External Interrupts on GPIO Pins

```c
#include <stdint.h>
#include "pico/stdlib.h"  // RP2040 standard library
#include "hardware/gpio.h"  // GPIO hardware access

// GPIO pin for the button
#define BUTTON_PIN 15

// Interrupt Service Routine (ISR) for the button press
void button_isr() {
    // Handle the interrupt event here
    printf("Button Pressed!\n");
}

int main() {
    stdio_init_all();  // Initialize stdio for printing (if using RP2040 SDK)

    gpio_init(BUTTON_PIN);
    gpio_set_dir(BUTTON_PIN, GPIO_IN);
    gpio_set_irq_enabled_with_callback(BUTTON_PIN, GPIO_IRQ_EDGE_FALL, true, &button

    // Enable interrupt for the button pin
    irq_set_exclusive_handler(IO_IRQ_BANK0); Interrupt type
    irq_set_enabled(IO_IRQ_BANK0, true);

    while (1) {
        // Your main application code can continue here
        // It will execute unless interrupted by the button press
    }

    return 0;
}
```



Button_isr is the callback function (interrupt handler) : functions whose address is passed in other functions to be called upon certain events.

Priority Levels handling

```c
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/irq.h"

void gpio_handler() {
    // Handle GPIO interrupt
}

int main() {
    stdio_init_all();

    gpio_init(PIN_NUMBER);
    gpio_set_dir(PIN_NUMBER, GPIO_IN);
    gpio_set_irq_enabled_with_callback(PIN_NUMBER, GPIO_IRQ_EDGE_RISE, true, &gpio_

    // Set GPIO interrupt priority (hypothetical function)
    irq_set_priority(IO_IRQ_GPIO0, 10); // Setting priority to 10 (example value)

    irq_set_exclusive_handler(IO_IRQ_GPIO0, gpio_handler);
    irq_set_enabled(IO_IRQ_GPIO0, true);

    // Main loop
    while (1) {
        // Your main application code here
    }

    return 0;
}
```

The function **irq_set_priority()** sets the priority of the GPIO0 interrupt to a numerical value of 10. This configuration would give the GPIO0 interrupt a mid-level priority.

THANK YOU