REPORT OF THE PROJECT OF ELECTRONIN FOR EMBEDED SYSTEM

supervisor: PROFESSOR Claudio Passerone

FIRST NAME: ALI

FAMILY NAME: ARAMNIA

Student number: 338956

COURSE: ELECTRONIN FOR EMBEDED SYSTEM

Department: Electronics and telecommunication

Politecnico di Torino

First semester of 2024/2025

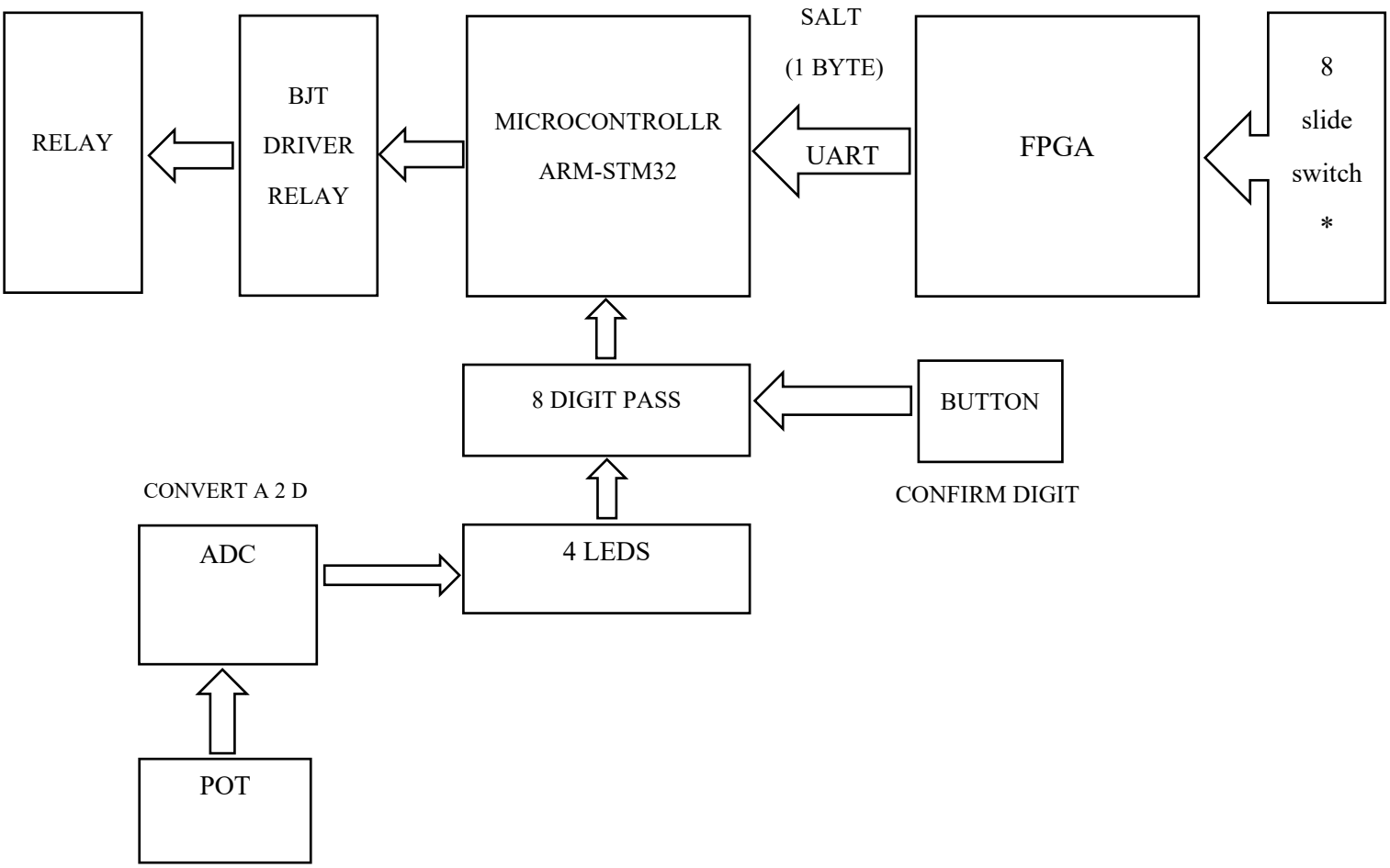
Project Index

Table of Contents

Contents

Outline

Secure Electronic Door Lock System Design



* 8 SLIDE SWITCH IS ON FPGA BOARD AND THEY ARE NOT APART

Overview

The project, "Secure Electronic Door Lock System Design," integrates hardware and software components to develop an efficient and secure access control system. The system utilizes an ARM-based STM32F103C8T6 microcontroller and a PYNQ-Z2 FPGA to manage user authentication, perform cryptographic operations, and control hardware peripherals such as LEDs, relays, and buttons.
Key highlights of the project include:

Hardware Integration:

Microcontroller: The STM32F103C8T6 handles data processing and interfaces with other peripherals using I2C and UART communication protocols.
FPGA: The PYNQ-Z2 board is used to implement a state machine for UART transmission, supporting efficient and reliable data communication.
Peripherals: Components such as ADC modules, LEDs, and relays are used to manage user input and provide feedback on system status.

Communication Protocols:

UART: Enables bidirectional data exchange between the FPGA and microcontroller, ensuring seamless interaction between components.
I2C: Used to connect the microcontroller with ADC modules, facilitating sensor data acquisition and control.

Software Functionality:

Password Management: The system allows users to enroll and store passwords securely in flash memory using SHA-256 hashing for enhanced security.
User Authentication: The system verifies user input against stored credentials, employing cryptographic techniques to ensure data integrity.
Real-time Feedback: Visual indicators (LEDs) and control mechanisms (relays) provide immediate feedback and system control.

Design and Implementation:

VHDL for FPGA: Implements a finite state machine (FSM) to manage UART data transmission, ensuring robust communication.
C Programming for STM32: Handles user input, cryptographic operations, and peripheral interfacing, ensuring a seamless user experience.
This project demonstrates an innovative approach to integrating embedded systems and secure communication technologies to create a practical and reliable electronic door lock system.

Peripheral Interfacing

- **Data Transmission**

In the context of interface connections, series and parallel describe how devices or components are connected to communicate with each other. Both terms apply to different types of data transmission and how systems interact, but they are used in slightly different ways depending on the type of system.

1. Series Interface Connection:

In a series connection, data or signals are transferred sequentially, one bit at a time, over a single communication line or channel. Devices are connected in a sequence, meaning that data moves from one device to the next in order.
In serial communication, data is transmitted over one wire, with the information being sent bit by bit in a continuous stream. Examples include protocols like UART, SPI, USB, and RS-232.
Example: A typical serial port communication like RS-232 uses a single wire to transmit data sequentially from one device to another.

Advantages:

Requires fewer wires, reducing complexity.
More efficient for long-distance communication because it's less susceptible to interference.

Disadvantages:

Slower speed compared to parallel communication, since data is sent one bit at a time.

2. Parallel Interface Connection:

In a parallel connection, data is transmitted simultaneously across multiple communication lines or channels. This means that multiple bits of data can be sent at the same time, usually on separate wires or pins.
In parallel communication, multiple data bits are transferred at once, allowing for faster data transmission. Examples include PCI, IDE (used for hard drives), and parallel printers.
Example: A parallel printer port (LPT port) used in older computers transferred data across multiple bits (usually 8 bits) at once using multiple wires.

Advantages:

Faster than serial communication because multiple bits are transmitted at once.
Suitable for short-distance communication where speed is essential.

Disadvantages:

Requires more wires, making the connection more complex.
Susceptible to interference and signal degradation over longer distances.

Series Interface Connection

When discussing series connections in the context of data communication protocols like SPI, I2C, and UART, we are essentially talking about how data is transmitted between devices. These protocols are often used for communication in embedded systems, microcontrollers, and other digital electronics. Let's break down how each of these protocols works in terms of data transmission (series connection) and their characteristics:

SPI (Serial Peripheral Interface):

Definition: SPI is a serial communication protocol that allows data to be transferred one bit at a time over a single data line (in a series fashion) between a master device and one or more slave devices. It uses multiple lines for communication (but still operates in series for bit-wise data transfer).
Series Connection Concept:
Master-Slave Communication: The master device sends out data bit by bit to the slave device(s).
Multiple Data Lines: While the data is transmitted serially (one bit at a time), SPI uses more than one line, including:

MOSI (Master Out Slave In): For sending data from master to slave.
MISO (Master In Slave Out): For sending data from slave to master.
SCK (Serial Clock): Used to synchronize data transfer.
SS (Slave Select): To select which slave device is being addressed.
Application: Typically used for high-speed communication between a microcontroller and peripherals like sensors, displays, and SD cards.

Data Transmission: The data is transferred serially, but SPI is considered a "parallel serial" system due to the use of multiple lines for separate functions (data, clock, and select lines).

I2C (Inter-Integrated Circuit):

Definition: I2C is another serial communication protocol but differs from SPI in how it connects devices. It uses only two wires for communication:
SDA (Serial Data Line): Carries the data.
SCL (Serial Clock Line): Carries the clock signal to synchronize the data transfer.
Series Connection Concept:

Master-Slave Communication: One master device communicates with multiple slave devices on the same bus.

Single Data Line: Data is transferred one bit at a time in series over the SDA line, with the clock signal provided by the master on the SCL line.
Addressing: Each device on the I2C bus has a unique address, and the master can communicate with any device by sending its address first.
Application: Common in connecting multiple devices with limited pin use, such as sensors, EEPROMs, and microcontrollers, within a system that doesn't need high-speed communication.
Data Transmission: Data is transferred serially (bit by bit) along the SDA line, with the clock signal synchronizing the transmission.

UART (Universal Asynchronous Receiver/Transmitter):

Definition: UART is a serial communication protocol that is used for asynchronous data transmission. It sends and receives data bit by bit over a single line in one direction at a time, typically with two main lines:

TX (Transmit): Transmits data from the transmitting device.
RX (Receive): Receives data at the receiving device.
Series Connection Concept:
Asynchronous Communication: Unlike SPI and I2C, UART does not require a clock signal to synchronize communication. Instead, it uses start and stop bits to frame the data, making it "asynchronous."
Full-Duplex: UART communication is typically full-duplex, meaning data can be sent and received simultaneously, but still, each bit is transferred one by one in series.
Application: UART is widely used in communication between computers and peripherals (e.g., GPS, sensors, Bluetooth modules), or between two microcontrollers.
Data Transmission: Data is sent bit by bit in a series, framed by start and stop bits to indicate the beginning and end of each data packet.

Series Interface Connection in our project

In our project, the ADC utilizes an I2C connection, while the communication between the FPGA and the microcontroller is established via UART.

I2C

In I2C (Inter-Integrated Circuit) communication, data transfer is structured with specific sequences involving addressing and acknowledgment mechanisms. Here’s a breakdown of how 7-bit addressing, n-bit addressing, data, and acknowledge (ACK) work in the I2C protocol:

1. 7-bit Addressing:

7-bit Address: I2C devices are addressed using a 7-bit address, allowing for 128 unique addresses (0 to 127). This is the most common type of addressing in I2C communication.

Format: The 7-bit address is transmitted first, and it is followed by a read/write bit to specify whether the master wants to read from or write to the slave device.

Example: If the address of the device is 0x50 (which is 0101 0000 in binary), the address part of the communication would be 0101000.

Address + R/W Bit: The 7-bit address is typically followed by an additional 1 bit indicating the read (R) or write (W) operation:
Write Operation: A 0 in the 8th bit indicates that the master will write data to the slave.
Read Operation: A 1 in the 8th bit indicates that the master will read data from the slave.

Example:
If the 7-bit address is 0x50, the address transmitted on the bus for a write operation would be 0xA0 (binary: 01010000 with 0 for write).
For a read operation, it would be 0xA1 (binary: 01010000 with 1 for read).

2. n-bit Addressing (10-bit Addressing):
10-bit Address: Some devices in I2C support a 10-bit addressing scheme, which allows for 1024 unique addresses. This is used when more than 128 devices need to be addressed on the bus.
Format: The 10-bit address is transmitted using a different sequence, with the first 7 bits identifying the 7-bit address of the master or the higher bits of the 10-bit address, and the next 3 bits containing the rest of the 10-bit address.
Example: The 10-bit address is split into two parts, with the first byte containing the upper 7 bits and the second byte containing the lower 3 bits.
Address + R/W: Similar to 7-bit addressing, the last bit will indicate whether the operation is a read or write.

3. Data Transfer:
After the address is sent, the master or slave will transfer data in a series of 8-bit chunks. Each byte of data is followed by an acknowledgment bit.
The data is usually transmitted in sequences of 8 bits (1 byte), where each byte represents a piece of information.
Write Operation: The master sends data to the slave device.
Read Operation: The master receives data from the slave device.

4. Acknowledge (ACK):
ACK Bit: After each byte of data is transmitted (whether it's an address or data), an acknowledgment (ACK) bit is sent by the receiving device (either the master or the slave).
ACK: A low (0) signal is sent as an acknowledgment, indicating that the receiver has successfully received the byte.
NACK: A high (1) signal is sent as a negative acknowledgment, indicating that the receiver has not successfully received the byte, or it signifies the end of communication (for example, after the last byte is received).
The ACK mechanism ensures reliable communication, as both devices confirm that the data has been received correctly.

Process:
After the master sends the 7-bit address + R/W bit, the slave device acknowledges by sending an ACK.
As the master sends each byte of data, the slave responds with an ACK.
The master can stop the transmission by sending a NACK at any point (for example, after receiving the last byte of data).

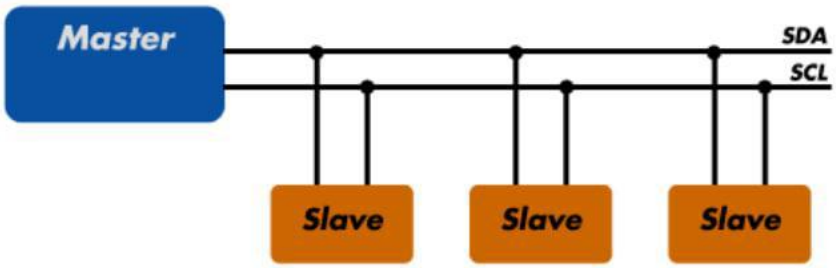
I2C Communication Flow (Step-by-Step):

1. Start Condition:
The master initiates the communication with a start condition (a high-to-low transition on the SDA line while SCL is high).

2. Addressing:
The master sends the 7-bit address of the slave device, followed by the R/W bit (to specify read or write).
The slave acknowledges (ACK) the address.

3. Data Transfer:
Data is transferred byte-by-byte between the master and slave.
After each byte of data, the receiving device sends an ACK or NACK.

4. Stop Condition:
The communication is terminated by the master issuing a stop condition (a low-to-high transition on the SDA line while SCL is high).

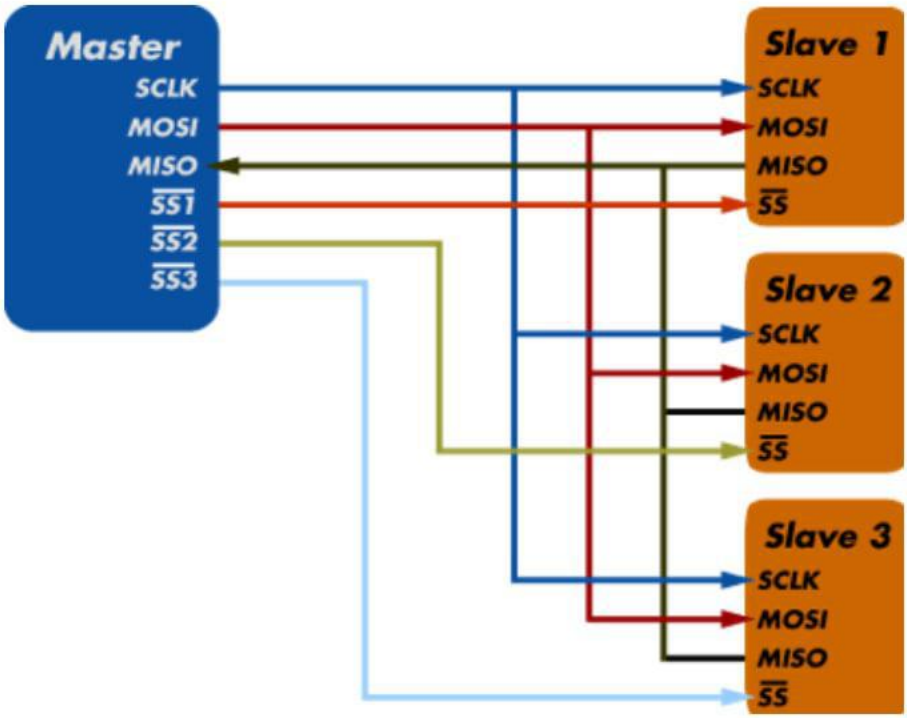


SPI

In SPI (Serial Peripheral Interface) communication, several signals are involved to facilitate the transfer of data between a master and one or more slave devices. You mentioned CS, CLK, DATA, and CS again, which are key components of the SPI protocol. Let's break down their roles:

1. CS (Chip Select or Slave Select):

Definition: CS (also known as SS - Slave Select) is a control signal used to select a specific slave device with which the master wants to communicate.
Function: When the CS line for a specific slave device is low (active), the slave is selected and can communicate with the master. If the CS line is high, the slave is deselected, and it will ignore any signals on the SPI bus.
Multiple Slaves: In systems with multiple slave devices, each slave will have its own CS line. The master selects the active slave by pulling the corresponding CS line low.



Each slave device requires a separate slave select signal (SS).

UART Data Flow and Baud Rate

UART (Universal Asynchronous Receiver/Transmitter) is a widely used communication protocol for transmitting and receiving serial data. It is asynchronous, meaning it doesn't require a clock signal to synchronize the communication between the transmitter and receiver. Instead, the devices involved must agree on a specific communication speed, defined by the baud rate.

1. Data Flow in UART:
The data flow in UART involves the transmission of data in the form of data frames. Each frame consists of several components that are transmitted sequentially, starting from the start bit to the stop bit(s).
Components of a UART Frame:

1. Start Bit: The transmission begins with a start bit (logical low or 0). This tells the receiver that data transmission is about to begin. The start bit is always a single bit.

2. Data Bits: The actual data that is transmitted. Typically, 8 bits (1 byte) of data are sent, but some configurations can use 5, 6, or 7 bits of data. The data bits are sent least significant bit (LSB) first.

3. Parity Bit (Optional): A parity bit may be included for error detection. The parity bit can be set to:
Even parity: Ensures an even number of 1's in the data bits and parity bit combined.
Odd parity: Ensures an odd number of 1's in the data bits and parity bit combined.
No parity: No error checking via parity.

4. Stop Bit(s): The transmission ends with one or more stop bits (usually 1 bit). The stop bit signals the end of the data frame and returns the line to an idle state (logical high or 1). 1 stop bit is most common, but you can also have 2 stop bits depending on the configuration.
UART Frame Structure Example (with 8 data bits, no parity, and 1 stop bit):
| Start Bit | Data Bits (8 bits) | Parity (None) | Stop Bit |
| 0 | 10101010 | - | 1 |
Start Bit: 0 (indicates the beginning of transmission)
Data Bits: 8 bits of actual data (in this example, 10101010)
Stop Bit: 1 (indicates the end of the transmission)

Data Flow Steps:
1. Start Bit: The transmitting device pulls the TX (Transmit) line low, signaling the beginning of data transmission.
2. Data Bits:The transmitter sends the data bits serially on the TX line, one bit at a time, synchronized by the agreed baud rate.
3. Parity Bit (Optional): If used, the parity bit follows the data bits and is checked by the receiver to verify data integrity.
4. Stop Bit(s): The stop bit(s) signal the end of the data transmission and allow the receiver to process the received data.

5. Idle State: After the stop bit, the TX line returns to its idle state (logical high or 1) until the next transmission.

Baud Rate:
The baud rate is the speed at which data is transmitted or received over a communication channel, expressed in bits per second (bps). It determines how fast bits are sent or received.
How Baud Rate Affects UART Communication:
The baud rate specifies how quickly the bits will be transmitted over the UART interface. The transmitter and receiver must agree on the same baud rate for successful communication. If there is a mismatch in baud rates between the two devices, the receiver may not interpret the bits correctly.
The baud rate is typically set before communication starts, and both the transmitting and receiving devices need to be configured with the same baud rate.
Examples of Common Baud Rates:
9600 bps (low speed, commonly used in many devices like sensors and modems)
115200 bps (higher speed, used in more demanding applications such as data transfer over serial ports)
19200 bps, 38400 bps, 57600 bps (intermediate speeds)
1 Mbps, 2 Mbps, etc. (very high-speed communication, often used in specific applications)
How Baud Rate Works in UART:
The baud rate tells both the transmitter and receiver how often to sample the incoming data bits.
For example:
At 9600 baud, the receiver expects one bit every 1/9600 seconds (~104 microseconds).
At 115200 baud, the receiver expects one bit every 1/115200 seconds (~8.7 microseconds).
Example: If you have multiple slave devices, the master controls which device is active by manipulating the corresponding CS line (one for each slave).

2. CLK (Clock):
Definition: CLK (or SCK for Serial Clock) is the clock signal generated by the master device to synchronize data transmission between the master and slave.
Function: The CLK line controls the timing of the data exchange. It defines when data is valid on the data lines.
On each clock pulse, data is shifted on the data lines (MOSI or MISO).
The frequency of the CLK determines the speed of data transfer.
Polarity and Phase: The CLK signal has two configuration parameters:
Clock Polarity (CPOL): Defines whether the clock is idle high or low.
Clock Phase (CPHA): Defines whether data is sampled on the rising edge or falling edge of the clock signal.
Example: The clock signal may oscillate at a specific frequency (e.g., 1 MHz), and each clock pulse triggers the transfer of one bit of data.

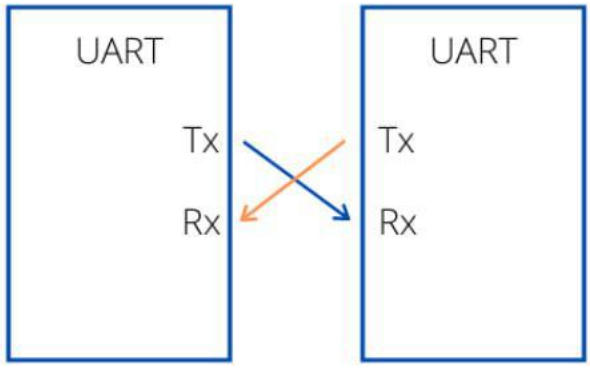
3. MOSI (Master Out Slave In):
Definition: MOSI is the data line used to transfer data from the master device to the slave device.
Function: As the clock pulses, data is shifted out from the master on the MOSI line, and the slave device receives the data.
Example: If the master wants to send a value to the slave (e.g., "0xA5"), it will shift the bits of 0xA5 onto the MOSI line, one bit per clock pulse.

4. MISO (Master In Slave Out):
Definition: MISO is the data line used to transfer data from the slave device to the master device.
Function: When the master requests data from the slave, the slave sends the data back on the MISO line. As the clock pulses, the slave shifts out its data onto the MISO line, and the master receives it.
Example: If the master wants to read data from the slave (e.g., the slave is sending "0x55"), the bits of 0x55 are shifted out of the slave onto the MISO line, one bit at a time, synchronized with the clock.

5. SPI Communication Overview:
SPI communication involves four main lines: CS, CLK, MOSI, and MISO. Data is transmitted serially between devices, with the clock signal ensuring synchronization.

Data Flow in SPI:

The master initiates communication by pulling the CS line low, selecting the slave device.
The master then generates clock pulses on the CLK line.
On each clock pulse, data is shifted out from the master to the slave on the MOSI line (and vice versa on the MISO line).
After the data transfer is complete, the CS line is pulled high to deselect the slave.
Full-Duplex Communication:
SPI supports full-duplex communication, meaning data can be transmitted and received simultaneously. The master sends data to the slave via the MOSI line while simultaneously receiving data from the slave via the MISO line.



UART Communication Definition

UART (Universal Asynchronous Receiver-Transmitter) is a method of asynchronous serial communication where data is transmitted between devices without the use of a shared clock signal. Instead, synchronization is achieved through specific signaling techniques and agreed-upon timing conventions.

1. Start Bit:
- Communication begins with the transmitter sending a start bit (a transition from logic high to logic low).
- The receiver detects this transition and synchronizes its internal clock to prepare for reading the incoming data.

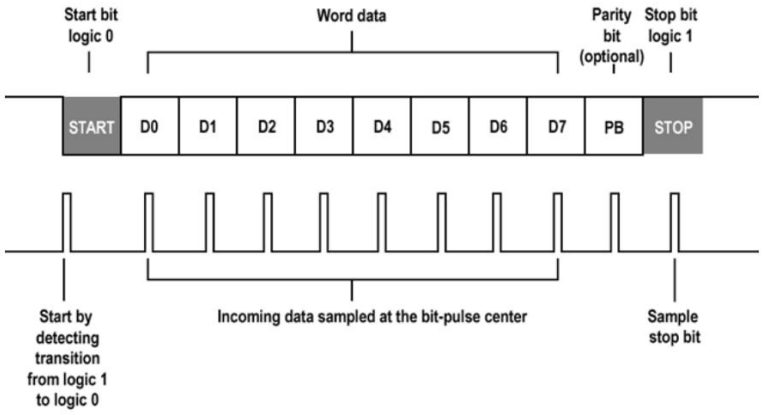
2. Data Bits (Word):
- The data being transmitted is referred to as the word.
- The word is sent one bit at a time, beginning with the least significant bit (LSB).
- Each bit is represented as a pulse of a specific duration (pre-determined by both devices). The receiver samples the line voltage at the midpoint of each bit interval to determine whether it is a logic 1 (high) or logic 0 (low).

3. Parity Bit (Optional):
- For basic error checking, a parity bit may be included.
- This bit is calculated based on the data bits to ensure the total number of logic 1s is either even (even parity) or odd (odd parity).

4. Stop Bit(s):
- At least one stop bit (a logic high) is transmitted at the end of each word.
- This signals the receiver that the transmission of the current word has concluded and allows it to reset for the next start bit.

5. Timing and Synchronization:
- Since no clock signal is transmitted, the devices must predefine the data rate (bits per second or baud rate).
- Both transmitter and receiver adhere to this agreed timing to ensure accurate data interpretation.

This mechanism enables UART communication to transmit data efficiently and reliably between devices with minimal wiring.



A UART frame

FSM in UART Transmitter

The UART transmitter FSM controls the process of sending data. The main states are:

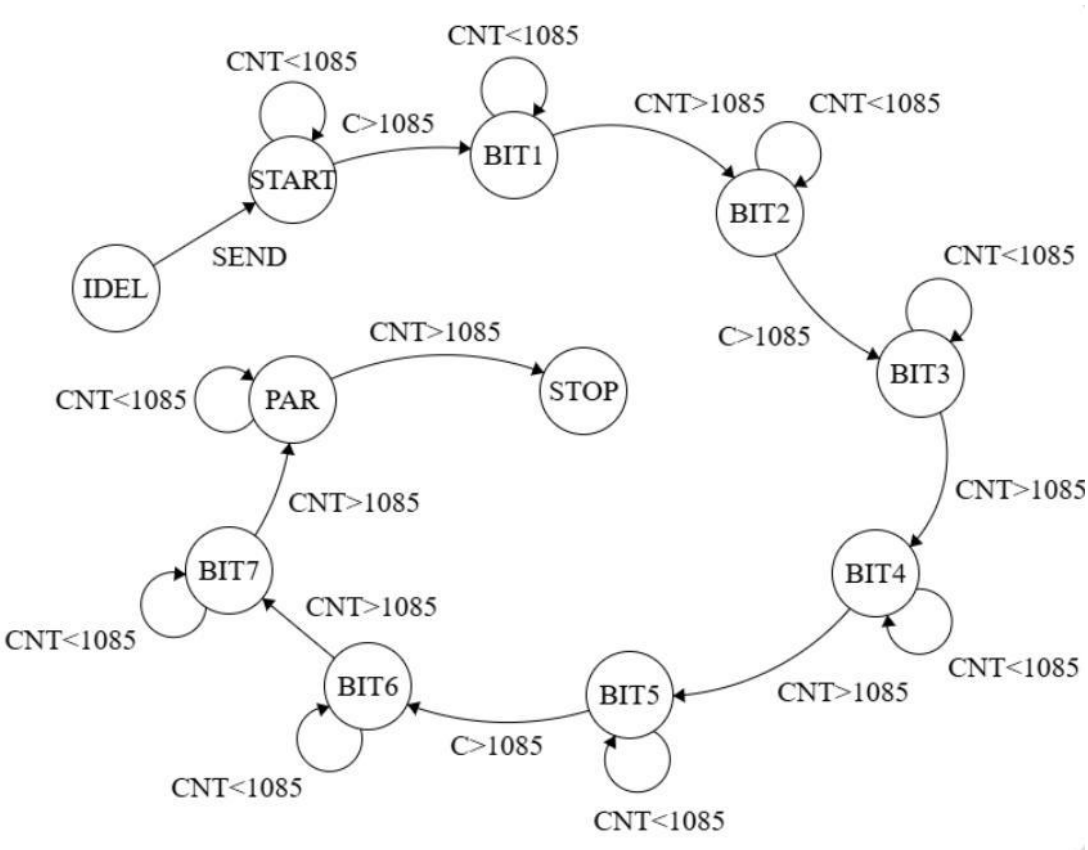
Idle State:
The transmitter is idle and waiting for data to be written to the transmit buffer.
Transition: If data is written, the FSM transitions to the Start state.

Start State:
The FSM sends the Start Bit (logic low).
Transition: After one clock cycle (or baud rate cycle), move to the Data state.

Data State:
The FSM transmits the **Data Bits** serially, one bit at a time.
Transition: After transmitting all data bits, move to the Parity or Stop state.

Parity State (Optional):
If parity is enabled, the FSM calculates and transmits the parity bit.
Transition: After sending the parity bit, move to the Stop state.

Stop State:
The FSM sends the **Stop Bit(s)** (logic high).
Transition: After the stop bit(s) are transmitted, return to the Idle state.



parity bit
the parity bit is used for error detection in the transmitted data. The parity bit is calculated so that the total number of 1 bits in the transmitted data (including the parity bit) is either even or odd, depending on the selected parity type:

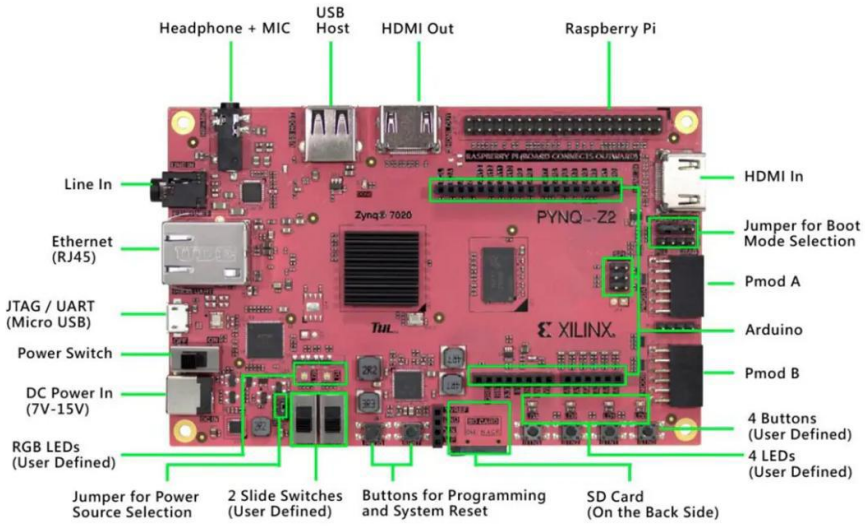
- Even parity: The total number of 1 bits must be even.
- Odd parity: The total number of 1 bits must be odd.

In the process, the number of 1 bits in the transmitted data is counted. Then, the parity bit is calculated and sent via the transmitter port along with the data. This allows the detection of single-bit errors in transmission.

7 bits of data	(count of 1-bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110

FPGA PYNQ-Z2

The **PYNQ-Z2** is a popular FPGA (Field Programmable Gate Array) development board designed for **embedded systems** and **machine learning** applications. It is built on the **Xilinx Zynq-7000 SoC (System on Chip)**, which combines a dual-core ARM Cortex-A9 processor with programmable logic (FPGA fabric)



The **PYNQ-Z2** is a popular FPGA (Field Programmable Gate Array) development board designed for **embedded systems** and **machine learning** applications. It is built on the **Xilinx Zynq-7000 SoC (System on Chip)**, which combines a dual-core ARM Cortex-A9 processor with programmable logic (FPGA fabric).
key features and details about the PYNQ-Z2 board:

Zynq-7000 SoC

Processor: Dual-core ARM Cortex-A9 (PS - Processing System).
FPGA: Artix-7 FPGA (PL - Programmable Logic).
Combines software programmability of ARM processors with hardware flexibility of FPGA.

Board Specifications

Programmable Logic Cells: ~85k.
Block RAM: ~4.9 Mb.
DSP Slices: 220.
Clock Speed: Up to 650 MHz for the FPGA fabric.

Memory

SDRAM: 512 MB DDR3 (shared between ARM processor and FPGA).
MicroSD Card Slot**:** For booting the PYNQ image and data storage.

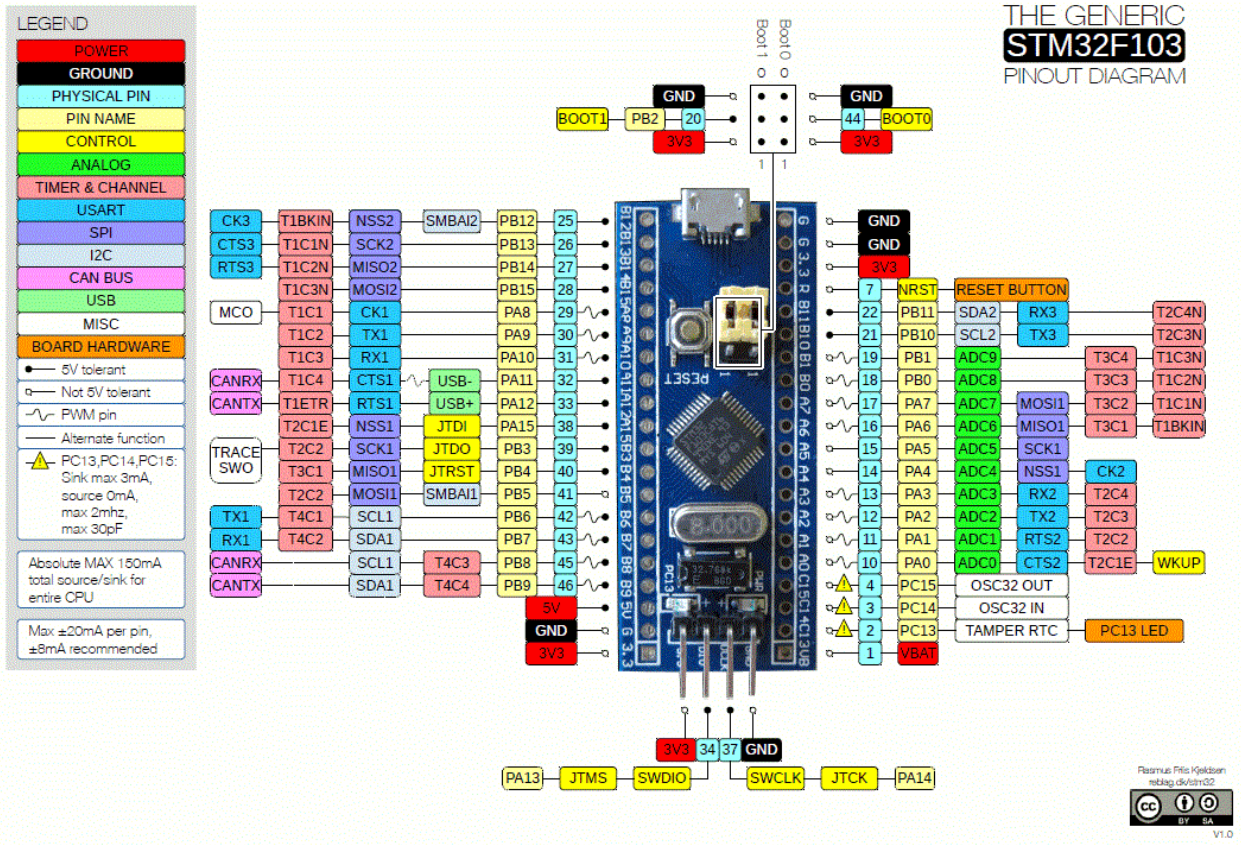
Connectivity

Ethernet: 1 Gbps Ethernet port.
USB Ports:
1 USB-UART (for programming/debugging).
1 USB OTG (for data transfer).

HDMI:
Input: 1 HDMI-IN for capturing video.
Output: 1 HDMI-OUT for video display.
Audio: Audio Codec for audio input/output.
PMOD Connectors: 2 PMOD headers for external peripherals.
Arduino Headers: For Arduino shields and custom extensions.

microcontroller stm32F103C8T6

The STM32F103C8T6 is a high-performance, low-cost microcontroller from STMicroelectronics' STM32F1 series, based on the ARM Cortex-M3 core. It is widely used in embedded systems due to its balance of processing power, peripheral support, and affordability.



- ## Key Features

Core:
ARM Cortex-M3, operating at up to 72 MHz.
32-bit architecture with efficient performance.

Memory:
64 KB Flash memory for program storage.
20 KB SRAM for data storage.

Peripherals:
USART/UART: Up to 3 interfaces for serial communication.
I2C: 2 interfaces for connecting devices like sensors or ADCs.
SPI: 2 interfaces for high-speed peripheral communication.
ADC: 10-bit, 16-channel ADC for analog-to-digital conversion.
Timers: Multiple general-purpose and advanced timers.

GPIO:
Up to 37 GPIO pins, configurable for digital input/output or alternate functions.
Communication:
Supports I2C, SPI, UART, and CAN for versatile communication needs.
Power:
Operates at 2.0V to 3.6V, making it compatible with a variety of systems.

Programmer Connection

The programmer connects to the STM32F103C8T6 microcontroller using 4 connections for programming and debugging via the SWD (Serial Wire Debug) interface:
SWDIO (Serial Wire Debug Input/Output): Data line for debugging and programming.
SWCLK (Serial Wire Clock): Clock signal for synchronizing the communication.

VCC (Voltage): Supplies power to the programmer, usually matching the microcontroller's voltage (e.g., 3.3V or 5V).
GND (Ground): Common ground connection between the programmer and the microcontroller.

Microcontroller to FPGA Communication (UART):

The communication between the STM32F103C8T6 and the FPGA uses the UART (Universal Asynchronous Receiver/Transmitter) protocol. The connections are:
TX (STM32) → RX (FPGA): Data sent from STM32 to FPGA.
RX (STM32) → TX (FPGA): Data received by STM32 from FPGA.
GND (Ground): Common ground for both devices.
This UART setup enables bidirectional serial communication between the microcontroller and FPGA.

Microcontroller to ADC (I2C Communication):

Connecting the PCF8574 (I/O Expander) Module with STM32 via I2C.

Overview of the PCF8574 Module

The PCF8574 module is an I/O expander that uses the I²C protocol to communicate with a microcontroller. This module is designed to increase the number of input/output pins of the microcontroller and is ideal for projects requiring multiple connections.

Using PCF8574 with ADC

Analog Inputs (ADC)

The PCF8574 module can interface with an ADC to receive analog data from sensors.
The ADC converts analog signals into digital values and transmits them to the microcontroller via the I²C protocol.

Digital Communication with ADC

The PCF8574 module acts as an intermediary, facilitating the transfer of digital signals from the ADC to the microcontroller over the I²C bus.

I²C Communication Protocol

Main Pins
SCL (Serial Clock): Transmits the clock signal.
SDA (Serial Data): Transmits and receives data.

Advantages
Reduces the number of wires required for communication.
Enables multiple I²C devices to connect to a shared bus.

Microcontroller Connection
SCL: Connected to the PB6 pin of the microcontroller.
SDA: Connected to the PB7 pin of the microcontroller.

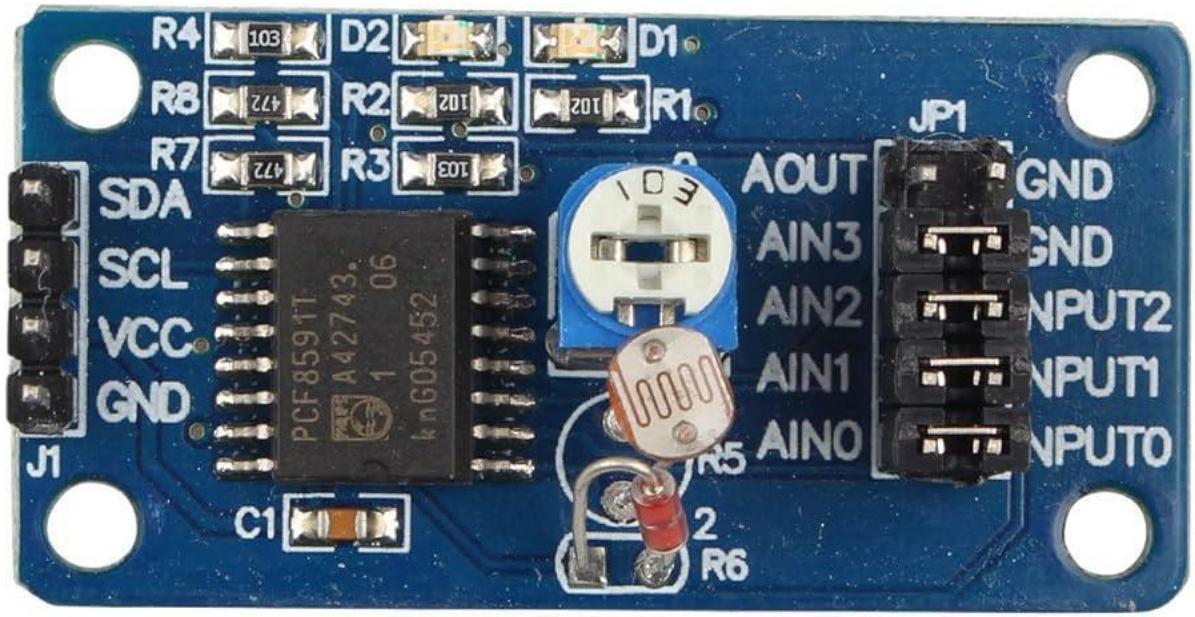
Pin Details

ADC Module Pins
AIN0 to AIN3: Four analog input channels for connecting sensors or analog signal sources (AIN3 is used in this project).
AOUT: Analog output produced by the DAC chip (not used in this project).

Module Connection Pins
GND: Connected to the circuit ground.
VCC: Supplies power to the module (3.3V in this project).
SCL (Serial Clock): Sends clock signals and is connected to PB6 of the microcontroller.
SDA (Serial Data): Sends and receives data and is connected to PB7 of the microcontroller.

Operation in the Project

Analog Signal Input
The desired analog signal is fed through the AIN3 channel of the ADC module.
The ADC converts the signal into a digital value proportional to the input voltage.
Connection to the Microcontroller
The I²C protocol facilitates communication between the ADC module and the microcontroller.
The SCL (PB6) and SDA (PB7) pins handle the clock and data transfer, respectively.
Power Supply and Ground
The module is powered via the VCC pin (3.3V).
The GND pin connects to the circuit ground.



LEDs for Displaying Digits:

The STM32 is connected to four LEDs representing a 4-bit binary value, with each pin corresponding to a bit:
PB15 (STM32) → MSB LED: Most significant bit.
PB14 (STM32) → Bit 2 LED.

PB13 (STM32) → Bit 1 LED.
PB12 (STM32) → LSB LED: Least significant bit.
Each LED can represent a binary digit, allowing visual representation of binary values.

Button with 2 Pins (Pull-Down Configuration)

In this project, the button has 2 pins connected as follows:
One pin of the button → VCC (3.3V): This pin is connected to the power supply and provides a high logic level when the button is pressed.
The other pin → Pull-Down resistor and GPIO pin (PA9): This pin is connected to the ground through a high-value Pull-Down resistor and is also connected to the PA9 GPIO pin of the microcontroller.

Relay Connection (Simulated with LED) and System Components

Relay Connection (Simulated with LED):

Instead of using a physical relay, LED with resistor connected to ground simulate the relay's behavior. The setup involves:

PA8 (STM32): Connected to the base of a **BJT transistor**, which acts as a switch to drive the LED.

BJT Transistor: Controls the LED switching mechanism. When activated by PA8, it allows current to flow through the LED, turning them on or off.

LEDs + Resistors: The LED are connected through resistors to ground, ensuring proper current regulation and mimicking relay operation.

This configuration allows the STM32 to efficiently simulate relay switching behavior using LED instead of mechanical relays.

System Components

BJT (Bipolar Junction Transistor):
The BJT acts as a switch.
When turned on, it allows current to flow from the collector to the emitter, completing the circuit and powering the door lock.
Relay:
An electrically controlled switch.
When the transistor is turned on, the relay closes, allowing current to flow through the lock coil (inductance).
Diode:

A diode is placed in parallel with the relay coil.
This **flyback diode** or **freewheeling diode** protects the transistor from voltage spikes caused by the inductance of the relay coil when the transistor turns off.
Inductance (Relay Coil):
The relay coil is an inductive load. When current flows through it, it creates a magnetic field that activates the lock mechanism.
This coil is powered when the transistor is on.

How It Works

Input Signal:
An input signal is sent to the base of the BJT (NPN or PNP, depending on configuration).
This signal turns the transistor on when it is high, allowing current to flow between the collector and emitter.

Transistor On:
When the transistor turns on (base is driven with a signal), the path between the collector and emitter is closed.
Current flows from the 5V power supply through the relay coil (inductance), activating the coil.
The relay coil creates a magnetic field that pulls the relay switch, closing the contacts and allowing current to flow to the door lock mechanism.

Flyback Diode Protection:
When the transistor turns off, the inductance of the relay coil tries to maintain current flow. This generates high voltage spikes.
The diode in parallel with the relay coil provides a safe path for this current to dissipate, preventing high-voltage spikes that could damage the transistor.
The diode allows current to flow through the coil and itself, clamping the voltage and protecting the circuit.

Key Points

Relay Control:
The relay is activated when the transistor is turned on, allowing current to flow through the relay coil and energizing the lock.

Flyback Diode:
The flyback diode is essential for protecting the transistor from voltage spikes caused by the inductive nature of the relay coil.

Inductive Load Behavior:
When the transistor turns off, the relay coil resists the sudden change in current, causing a voltage spike. The flyback diode safely absorbs this energy.

Circuit Explanation:

Input Signal:
The input signal originates from a microcontroller and is fed to the base of the BC547 transistor through a 1kΩ resistor.
This resistor limits the base current to protect the transistor.

Transistor Operation:
When the input signal is HIGH, current flows into the base of the BC547, turning the transistor ON.
When the transistor is ON, it completes the circuit for the relay coil, allowing current to flow from the 5V supply through the coil to ground.

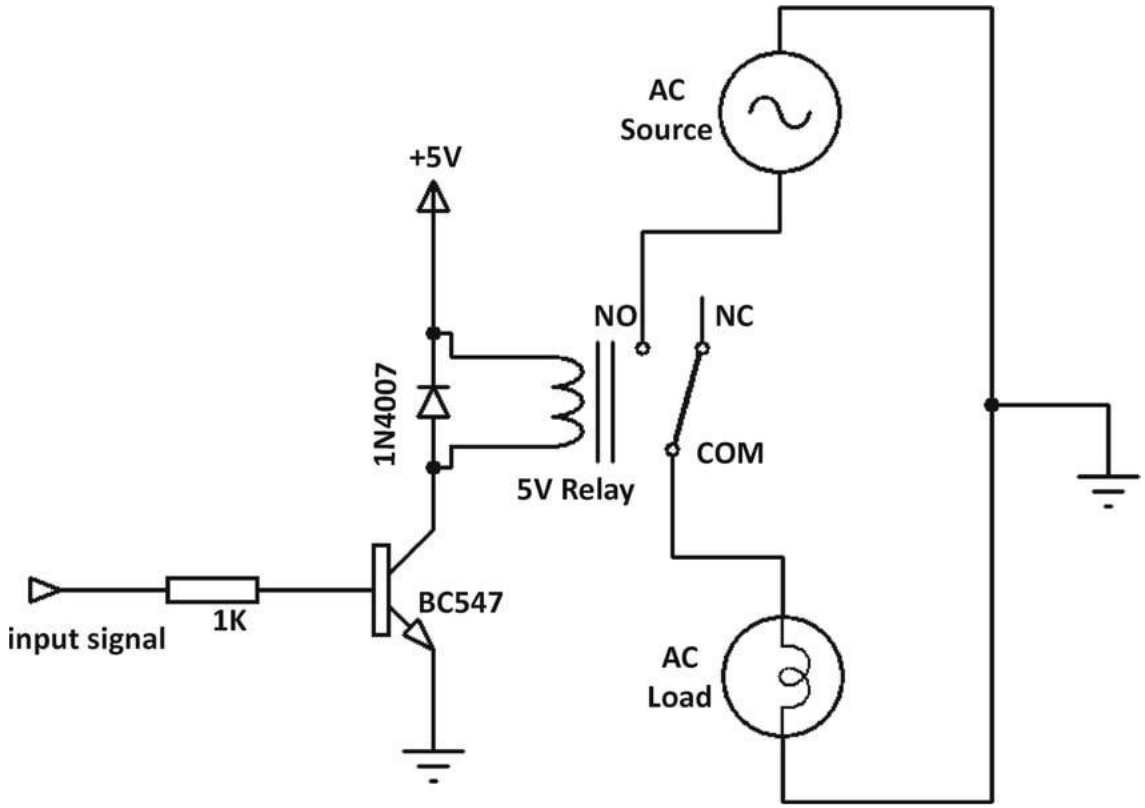
Relay Operation:
The 5V relay is energized when the transistor is ON. This changes the state of the relay:
The Normally Open (NO) terminal connects to the Common (COM) terminal.
This allows the AC Source to power the AC Load.

Diode (1N4007):
The 1N4007 diode is placed across the relay coil in a reverse-biased configuration.
Its purpose is to suppress voltage spikes generated by the inductive coil when the relay is turned OFF (flyback voltage).
This protects the transistor from damage.

AC Load Control:
The relay is used to control the AC load, which could be a light, motor, or any other AC appliance.
The AC source is connected to the COM terminal, and the AC load is connected to the NO terminal. The load is powered only when the relay is energized.

Transistor OFF State:
When the input signal is LOW, the base of the BC547 receives no current, turning the transistor OFF.
This de-energizes the relay, causing the NO terminal to disconnect from the COM terminal, and the AC load is turned OFF.

Simulation with LED
In some cases (e.g., for testing purposes), the relay's behavior can be simulated using an LED and a current-limiting resistor. When the transistor is ON, the LED will light up, indicating the relay activation.



This allows the STM32 to control the LED, simulating relay functionality visually.

Summary of Connections:

Programming/Debugging:
SWDIO → Debugger/Programmer
SWCLK → Debugger/Programmer
VCC → Power supply
GND → Ground

Microcontroller to FPGA (UART):
TX (STM32) → RX (FPGA)
RX (STM32) → TX (FPGA)

Microcontroller to ADS (I2C):
PB6 (STM32) → SCL (ADS)
PB7 (STM32) → SDA (ADS)

Relay Simulation (with LED):
PA8 (STM32) → LED anode (with resistor to ground)

4 LEDs for Digit Representation:
PB15 (STM32) → MSB LED
PB14 (STM32) → Bit 2 LED
PB13 (STM32) → Bit 1 LED
PB12 (STM32) → LSB LED

Button (Pull-down Configuration):
One Leg → VCC (Power Supply)
Other Leg → GPIO Pin (STM32) (e.g., PA0)
Third Leg → Ground

CODE SECTION

VIVADO(VHDL)

```
library IEEE;                               -- Standard IEEE library for logic
and types
use IEEE.STD_LOGIC_1164.ALL;                -- Standard logic 1164 library for
std_logic type and related operations

entity Transmitter is                       -- Declare an entity named
"Transmitter"
Port (
    clock: in std_logic;                        -- Clock input signal of type
std_logic
    transmitter: out std_logic;                 -- Output signal to send data
    send_data: in std_logic;                    -- Input signal indicating data
should be sent
    slide: in std_logic_vector(1 downto 0)    -- A 2-bit std_logic_vector
);
end Transmitter;

architecture Behavioral of Transmitter is
    -- Define an enumeration type for the UART transmitter FSM states
    type UART_transmitter is (idle, start, bit0, bit1, bit2, bit3, bit4,
bit5, bit6, bit7, parity, stop);
    signal tx_fsm: UART_transmitter := idle;   -- FSM signal for UART
transmitter, initialized to idle
```

```
    signal baud_rate_counter: integer := 0;    -- Counter to track baud rate
timing
    signal TX_Data: std_logic_vector(7 downto 0) := x"AA"; -- 8-bit data to
be transmitted
    signal index: integer := 0;                -- Index to iterate over bits
of TX_Data
    signal parity_bit: std_logic;              -- Calculated parity bit
    signal parity_type: std_logic := '0';      -- Parity type: '0' for even
parity, '1' for odd parity

begin
    -- Process to calculate the parity bit based on TX_Data and parity_type
    process(TX_Data, parity_type)
        variable count: integer := 0;          -- Local variable to count '1'
bits in TX_Data
    begin
        count := 0;                            -- Reset count
        for i in 0 to 7 loop                   -- Loop through bits of
TX_Data
            if TX_Data(i) = '1' then
                count := count + 1;            -- Increment count if bit is
'1'
            end if;
        end loop;

        -- Set parity_bit based on parity_type
        if parity_type = '0' then              -- Even parity
            if (count mod 2) = 0 then
                parity_bit <= '0';             -- Even count results in
parity bit '0'
            else
                parity_bit <= '1';             -- Odd count results in parity
bit '1'
            end if;
        else                                   -- Odd parity
            if (count mod 2) = 0 then
                parity_bit <= '1';             -- Even count results in
parity bit '1'
            else
                parity_bit <= '0';             -- Odd count results in parity
bit '0'
            end if;
        end if;
    end process;

    -- Main process that updates the FSM on the rising edge of the clock
    process(clock)
    begin
        if rising_edge(clock) then            -- Trigger on the rising edge
of the clock
```

```
            case tx_fsm is                    -- Evaluate FSM state
                when idle =>
                    baud_rate_counter <= 0;   -- Reset baud counter
                    transmitter <= '1';       -- Keep idle state high
                    index <= 0;               -- Reset index
                    if send_data = '1' then   -- Move to start state when
send_data is asserted
                        tx_fsm <= start;
                    end if;

                when start =>
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= '0';       -- Transmit start bit
                    if baud_rate_counter = 1085 then
                        baud_rate_counter <= 0;
                        tx_fsm <= bit0;       -- Move to bit0 state
                    end if;

                when bit0 =>
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= slide(index);  -- Transmit data from slide
vector
                    if baud_rate_counter = 1085 then
                        baud_rate_counter <= 0;
                        tx_fsm <= bit1;
                        index <= index + 1;
                    end if;

                when bit1 =>
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= slide(index);  -- Transmit next bit
                    if baud_rate_counter = 1085 then
                        baud_rate_counter <= 0;
                        tx_fsm <= bit2;
                        index <= index + 1;
                    end if;

                when bit2 =>
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= TX_Data(index); -- Transmit actual TX_Data
bits
                    if baud_rate_counter = 1085 then
                        baud_rate_counter <= 0;
                        tx_fsm <= bit3;
                        index <= index + 1;
                    end if;

                when bit3 =>
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= TX_Data(index);
```

```
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= bit4;
        index <= index + 1;
    end if;

when bit4 =>
    baud_rate_counter <= baud_rate_counter + 1;
    transmitter <= TX_Data(index);
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= bit5;
        index <= index + 1;
    end if;

when bit5 =>
    baud_rate_counter <= baud_rate_counter + 1;
    transmitter <= TX_Data(index);
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= bit6;
        index <= index + 1;
    end if;

when bit6 =>
    baud_rate_counter <= baud_rate_counter + 1;
    transmitter <= TX_Data(index);
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= bit7;
        index <= index + 1;
    end if;

when bit7 =>
    baud_rate_counter <= baud_rate_counter + 1;
    transmitter <= TX_Data(index);
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= parity;
    end if;

when parity =>
    baud_rate_counter <= baud_rate_counter + 1;
    transmitter <= parity_bit;
    if baud_rate_counter = 1085 then
        baud_rate_counter <= 0;
        tx_fsm <= stop;
    end if;

when stop =>
```

```
                    baud_rate_counter <= baud_rate_counter + 1;
                    transmitter <= '1';   -- Transmit stop bit
                    if baud_rate_counter = 1085 then
                        baud_rate_counter <= 0;
                        tx_fsm <= idle;   -- Return to idle state
                    end if;
            end case;
        end if;
    end process;
end Behavioral;
```

STM32(C LANGUAGE)

```
/* USER CODE BEGIN Header */
/**
******************************************************************************
*
* @file           : main.c
* @brief          : Main program body
******************************************************************************
*
* @attention
*
* Copyright (c) 2024 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is provided AS-IS.
*
******************************************************************************
*
*/
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------
-*/
#include "main.h"
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include "cmox_crypto.h"   // For cryptographic operations
#include "MY_FLASH.h"      // For Flash memory operations

/* Debugging Macro */
#define printf_debug 1 // Enable or disable debug printing
#if printf_debug
    #define DEBUG_PRINT(fmt, ...) printf(fmt, ##__VA_ARGS__)
```

```
#else
    #define DEBUG_PRINT(fmt, ...)
#endif

/* I2C Address and ADC Channel */
#define PCF8591 (0x48 << 1) // Address for the PCF8591 ADC module
#define ADC3 0x03           // ADC channel to read from

/* Function Prototypes */
uint8_t read_ADC3_value(I2C_HandleTypeDef *hi2c);
uint8_t mapToRange(uint8_t value);
void displayOnLEDs(uint8_t value);
void sha256(uint8_t *data, uint8_t *result, uint8_t DataSize);
bool Read_btn();
void enrollment();
bool unlock();
uint8_t password[8];
uint8_t saved_password[32];
void UART2_Init(void);
uint8_t UART2_ReceiveByte(void);

/* Private variables */
CRC_HandleTypeDef hcrc;
I2C_HandleTypeDef hi2c1;

/* Redirect printf to ITM Debug Console */
int _write(int file, char *ptr, int len) {
    (void)file;
    for (int DataIdx = 0; DataIdx < len; DataIdx++) {
        ITM_SendChar(*ptr++);
    }
    return len;
}

/* Scanning I2C addresses */
void I2C_Scan() {
    printf("Scanning I2C bus...\n");
    for (uint8_t i = 1; i < 128; i++) {
        if (HAL_I2C_IsDeviceReady(&hi2c1, (i << 1), 1, HAL_MAX_DELAY) ==
HAL_OK) {
            DEBUG_PRINT("Device found at 0x%02X\n", i);
        }
    }
    DEBUG_PRINT("Scan complete.\n");
}

/* Print array for debugging */
void printArray(const char *arrayName, uint8_t *array, size_t size) {
    DEBUG_PRINT("%s: ", arrayName);
    for (size_t i = 0; i < size; i++) {
```

```
        DEBUG_PRINT("0x%02X ", array[i]);
    }
    DEBUG_PRINT("\n");
}

/* Application entry point */
int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_CRC_Init();
    UART2_Init();
    HAL_GPIO_WritePin(builtIn_led_GPIO_Port, builtIn_led_Pin, 1);
    DEBUG_PRINT("Hello world...\n");
    enrollment();
    DEBUG_PRINT("Enrollment is done\n");
    page_read(saved_password, sizeof(saved_password));
    while (1) {
        if (unlock()) {
            HAL_GPIO_WritePin(builtIn_led_GPIO_Port, builtIn_led_Pin, 0);
            DEBUG_PRINT("Correct\n");
        } else {
            DEBUG_PRINT("Wrong\n");
            HAL_GPIO_WritePin(builtIn_led_GPIO_Port, builtIn_led_Pin, 1);
        }
    }
}

/* Read ADC3 value */
uint8_t read_ADC3_value(I2C_HandleTypeDef *hi2c) {
    uint8_t control_byte = ADC3;
    uint8_t data[2] = {0};
    if (HAL_I2C_Master_Transmit(hi2c, PCF8591, &control_byte, 1,
HAL_MAX_DELAY) != HAL_OK) {
        return 0;
    }
    if (HAL_I2C_Master_Receive(hi2c, PCF8591, data, 2, HAL_MAX_DELAY) !=
HAL_OK) {
        return 0;
    }
    return data[1];
}

/* Map value to range */
uint8_t mapToRange(uint8_t value) {
    return value / 16;
}

/* Display on LEDs */
```

```
void displayOnLEDs(uint8_t value) {
    HAL_GPIO_WritePin(Bit0_GPIO_Port, Bit0_Pin, (value & 0x01) ? GPIO_PIN_SET
: GPIO_PIN_RESET);
    HAL_GPIO_WritePin(Bit1_GPIO_Port, Bit1_Pin, (value & 0x02) ? GPIO_PIN_SET
: GPIO_PIN_RESET);
    HAL_GPIO_WritePin(Bit2_GPIO_Port, Bit2_Pin, (value & 0x04) ? GPIO_PIN_SET
: GPIO_PIN_RESET);
    HAL_GPIO_WritePin(Bit3_GPIO_Port, Bit3_Pin, (value & 0x08) ? GPIO_PIN_SET
: GPIO_PIN_RESET);
}

/* Compute SHA-256 hash */
void sha256(uint8_t *data, uint8_t *result, uint8_t DataSize) {
    size_t computed_size;
    cmox_hash_compute(CMOX_SHA256_ALGO, data, DataSize, result,
CMOX_SHA256_SIZE, &computed_size);
}

/* Read button state */
bool Read_btn() {
    HAL_Delay(5);
    return HAL_GPIO_ReadPin(btn_GPIO_Port, btn_Pin);
}

/* UART2 Initialization */
void UART2_Init(void) {
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
    GPIOA->CRL &= ~GPIO_CRL_CNF2;
    GPIOA->CRL |= GPIO_CRL_CNF2_1 | GPIO_CRL_MODE2;
    GPIOA->CRL &= ~GPIO_CRL_CNF3;
    GPIOA->CRL |= GPIO_CRL_CNF3_0;
    GPIOA->CRL &= ~GPIO_CRL_MODE3;
    USART2->BRR = 138;
    USART2->CR1 |= USART_CR1_M | USART_CR1_PCE;
    USART2->CR1 &= ~USART_CR1_PS;
    USART2->CR2 &= ~USART_CR2_STOP;
    USART2->CR1 |= USART_CR1_TE | USART_CR1_RE | USART_CR1_UE;
    while (!(USART2->SR & USART_SR_TC));
}

/* UART2 Receive Byte */
uint8_t UART2_ReceiveByte(void) {
    while (!(USART2->SR & USART_SR_RXNE));
    return (uint8_t)(USART2->DR & 0xFF);
}
```