Silicon-Interconnect Communications

Register-Level I2C Implementation

A Deep-Dive into Hardware-Software Interface

VLSI Engineering Portfolio Project

February 5, 2026

Abstract

This document presents a comprehensive analysis of a bare-metal I2C communication system implemented on an STM32 microcontroller. The project demonstrates mastery of digital logic design, protocol implementation, and hardware-software co-design by manually generating I2C signals at the register level. Through bit-banging techniques, this implementation controls an LCD display via a PCF8574 I/O expander without relying on hardware abstraction layers, providing deep insight into the electrical and logical behavior of silicon interconnects.

Contents

1 Introduction

1.1 Project Overview

The Silicon-Interconnect-Communications project represents a fundamental exercise in understanding the interface between software and silicon. Rather than utilizing pre-built libraries or hardware peripherals, this implementation manually generates the electrical signals required for I2C communication, demonstrating complete control over the hardware at the gate level.

1.2 Objectives

- Implement I2C protocol through direct register manipulation
- Understand timing requirements and signal integrity in serial communication
- Master the PCF8574 I/O expander interface protocol
- Achieve functional LCD control without hardware abstraction
- Demonstrate knowledge of VLSI concepts including atomicity, timing analysis, and bus arbitration

1.3 System Architecture

The project implements a four-layer architecture stack:

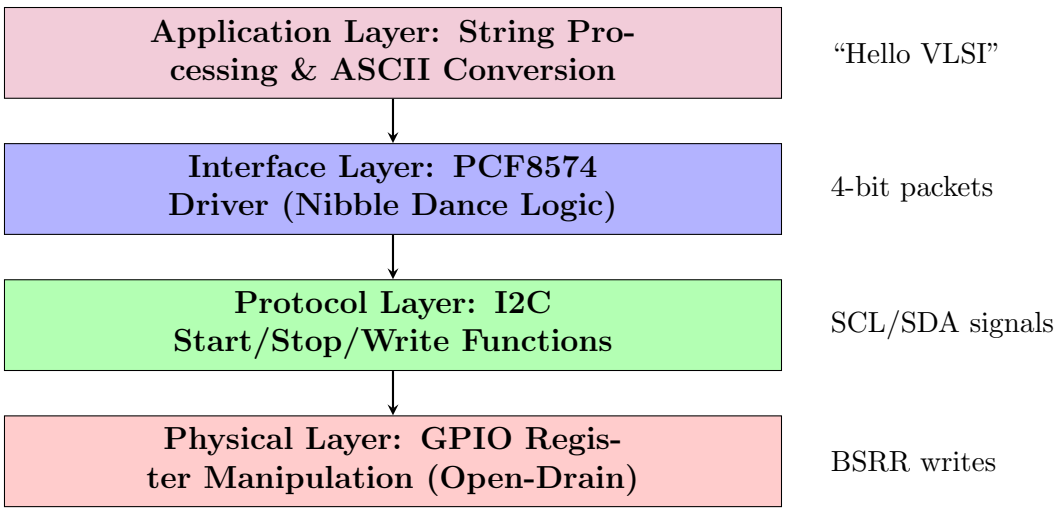


Figure 1: Four-Layer Architecture Stack

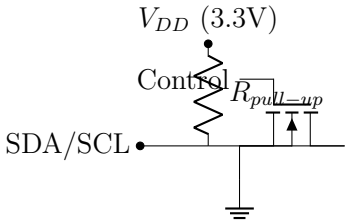
2 Hardware Configuration

2.1 GPIO Open-Drain Configuration

The STM32 GPIO pins are configured in **Open-Drain** mode, which is essential for I2C communication. This configuration allows multiple devices to share the same bus without electrical contention.

Open-Drain Characteristics

- Output can pull line LOW (active drive)
- Output cannot drive HIGH (passive, relies on pull-up resistor)
- Prevents short circuits when multiple devices access the bus
- Implements wired-AND logic



Control = 0 → NMOS OFF → Line HIGH
Control = 1 → NMOS ON → Line LOW

Figure 2: Open-Drain Output Configuration

2.2 Register-Level Pin Control

The project uses the **BSRR** (Bit Set/Reset Register) for atomic pin manipulation:

$$\text{BSRR}[15:0] = \text{Set bits (write 1 to set pin HIGH)} \tag{1}$$

$$\text{BSRR}[31:16] = \text{Reset bits (write 1 to set pin LOW)} \tag{2}$$

Atomicity Guarantee

BSRR allows changing pin states in a single CPU clock cycle, preventing race conditions and signal glitches that could corrupt I2C communication.

```
#define SCL_HIGH()  SCL_GPIO_Port->BSRR = SCL_Pin
#define SCL_LOW()   SCL_GPIO_Port->BSRR = (uint32_t)SCL_Pin << 16
#define SDA_HIGH()  SDA_GPIO_Port->BSRR = SDA_Pin
#define SDA_LOW()   SDA_GPIO_Port->BSRR = (uint32_t)SDA_Pin << 16
#define SDA_READ()  (SDA_GPIO_Port->IDR & SDA_Pin)
```

Listing 1: Pin Control Macros

3 I2C Protocol Implementation

3.1 I2C Bus Fundamentals

I2C (Inter-Integrated Circuit) is a synchronous, half-duplex, serial communication protocol using two wires:

- **SCL** (Serial Clock): Clock signal generated by master
- **SDA** (Serial Data): Bi-directional data line

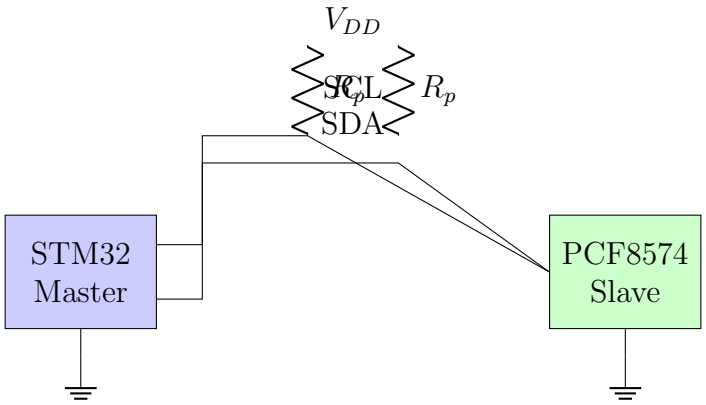


Figure 3: I2C Bus Topology with Pull-up Resistors

3.2 I2C Timing Specification

Parameter	Symbol	Standard Mode
SCL Clock Frequency	f_{SCL}	100 kHz
SCL Low Period	t_{LOW}	4.7 μs
SCL High Period	t_{HIGH}	4.0 μs
SDA Setup Time	$t_{su;DAT}$	250 ns
SDA Hold Time	$t_{hd;DAT}$	0 ns (min)
Start Condition Hold	$t_{hd;STA}$	4.0 μs
Stop Condition Setup	$t_{su;STO}$	4.0 μs

Table 1: I2C Timing Parameters (Standard Mode)

3.3 I2C Start Condition

The START condition signals the beginning of a transmission. It is defined as a HIGH-to-LOW transition on SDA while SCL is HIGH.

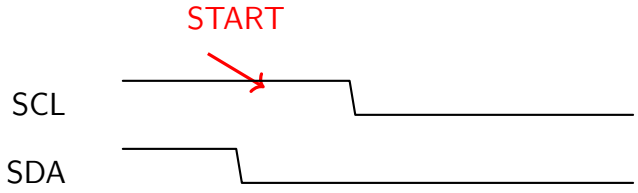


Figure 4: I2C Start Condition Timing

```c
void I2C_Start(void) {
    SDA_HIGH();  // Ensure SDA is HIGH
    SCL_HIGH();  // Ensure SCL is HIGH
    I2C_Delay(); // Wait for setup time
    SDA_LOW();   // Pull SDA LOW (START condition)
    I2C_Delay(); // Hold time
    SCL_LOW();   // Pull SCL LOW (ready for data)
    I2C_Delay();
}
```

Listing 2: Start Condition Implementation

3.4 I2C Stop Condition

The STOP condition signals the end of transmission. It is defined as a LOW-to-HIGH transition on SDA while SCL is HIGH.

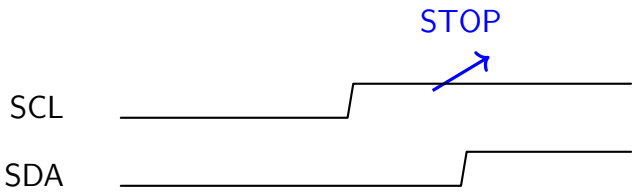


Figure 5: I2C Stop Condition Timing

```c
void I2C_Stop(void) {
    SDA_LOW();   // Ensure SDA is LOW
    I2C_Delay();
    SCL_HIGH();  // Release SCL (allow to go HIGH)
    I2C_Delay(); // Setup time
    SDA_HIGH();  // Release SDA (STOP condition)
    I2C_Delay(); // Bus free time
}
```

Listing 3: Stop Condition Implementation

3.5 I2C Data Transmission

Data is transmitted MSB (Most Significant Bit) first. Each byte is followed by an ACK (Acknowledge) bit from the receiver.

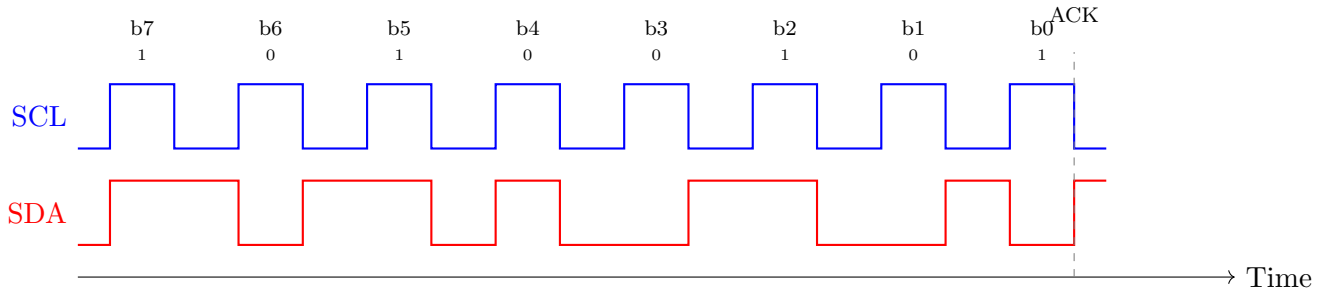


Figure 6: I2C Byte Transmission (0xA5 with ACK)

Key Timing Rules

1. SDA must be stable during SCL HIGH period
2. SDA can only change when SCL is LOW
3. Data is sampled on SCL rising edge
4. Violating these rules causes communication errors

```
void I2C_Write(uint8_t data) {
    for (int i = 0; i < 8; i++) {
        // Set SDA based on current bit (MSB first)
        if (data & (0x80 >> i))
            SDA_HIGH();
        else
            SDA_LOW();

        I2C_Delay();    // Setup time: SDA stable before SCL rise
        SCL_HIGH();     // Clock pulse: receiver samples data
        I2C_Delay();    // SCL high period
        SCL_LOW();      // Return clock to LOW
    }

    // ACK Pulse: Release SDA, slave pulls LOW if ACK
    SDA_HIGH();
    SCL_HIGH();
    I2C_Delay();
    SCL_LOW();
}
```

Listing 4: Byte Write Function

3.6 Complete I2C Transaction

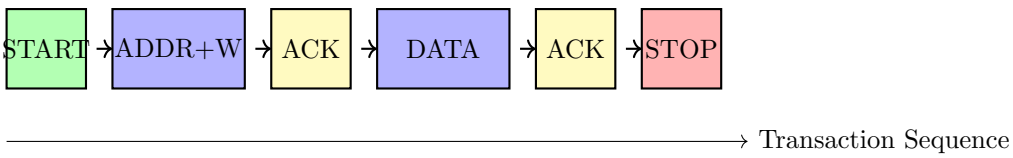


Figure 7: Complete I2C Write Transaction

4 PCF8574 I/O Expander Interface

4.1 Device Overview

The PCF8574 is an 8-bit I/O expander that provides 8 GPIO pins controlled via I2C. In this project, it serves as an interface between the STM32 and the LCD display.

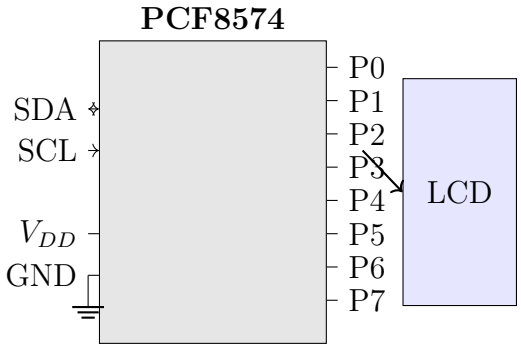


Figure 8: PCF8574 Pin Configuration and LCD Connection

4.2 Pin Mapping to LCD

PCF8574 Pin	Bit Position	LCD Signal	Function
P0	bit 0	RS	Register Select (0=Command, 1=Data)
P1	bit 1	RW	Read/Write (0=Write)
P2	bit 2	E	Enable (falling edge trigger)
P3	bit 3	BL	Backlight control
P4	bit 4	D4	Data bit 4
P5	bit 5	D5	Data bit 5
P6	bit 6	D6	Data bit 6
P7	bit 7	D7	Data bit 7

Table 2: PCF8574 to LCD Pin Mapping

```
#define LCD_ADDR (0x27 << 1)  // I2C address (left-shifted for R/W bit)
#define RS_BIT   (1 << 0)      // Register Select on P0
#define EN_BIT   (1 << 2)      // Enable on P2
#define BL_BIT   (1 << 3)      // Backlight on P3
```

Listing 5: Pin Mapping Definitions

4.3 The "Nibble Dance" - 4-bit Mode Operation

The LCD operates in 4-bit mode to save pins. Each 8-bit value must be sent as two 4-bit "nibbles" (high nibble first, then low nibble).

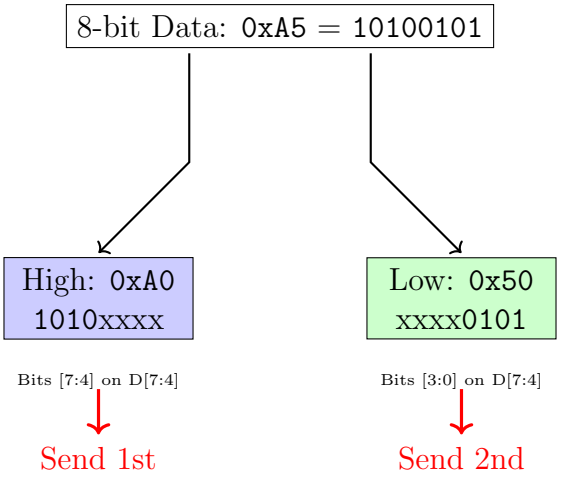


Figure 9: 4-bit Nibble Transmission Sequence

```
void LCD_Send(uint8_t val, uint8_t rs) {
    // Extract high and low nibbles
    uint8_t high = (val & 0xF0) | rs;          // Keep upper 4 bits
    uint8_t low = ((val << 4) & 0xF0) | rs;    // Shift lower 4 bits up

    // Send High Nibble with Enable pulse
    LCD_InternalWrite(high | EN_BIT);  // E=1: latch data
    I2C_Delay();
    LCD_InternalWrite(high);           // E=0: falling edge trigger

    // Send Low Nibble with Enable pulse
    LCD_InternalWrite(low | EN_BIT);   // E=1: latch data
    I2C_Delay();
    LCD_InternalWrite(low);            // E=0: falling edge trigger
}
```

Listing 6: Nibble Dance Implementation

4.4 Enable Pin Timing

The LCD's Enable (E) pin requires a specific pulse sequence. Data is latched on the **falling edge** of the Enable signal.

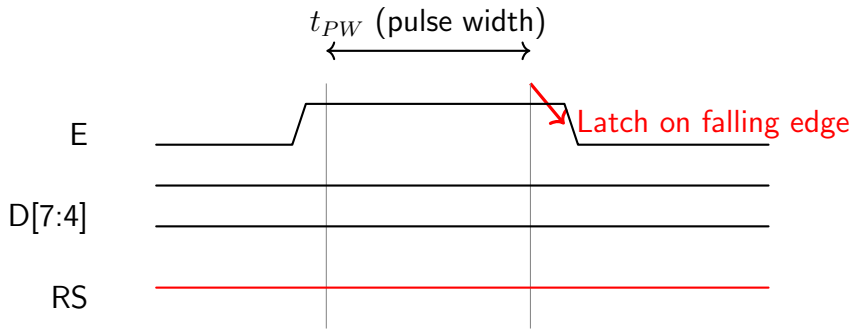


Figure 10: LCD Enable Pin Timing Diagram

5 Complete Data Flow Analysis

5.1 Character 'H' Transmission Example

Let's trace the complete transmission of the ASCII character 'H' (0x48) to the LCD.

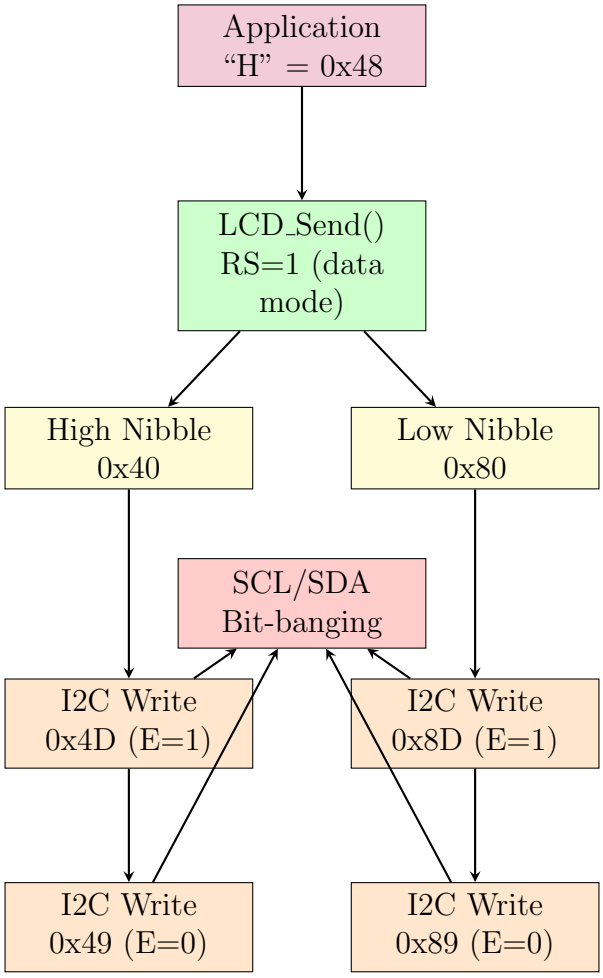


Figure 11: Complete Data Flow for Character 'H'

5.2 Byte Value Calculation

For character 'H' (0x48 = 0100 1000):

$$\text{High Nibble} = (\texttt{0x48}\&\texttt{0xF0})|\text{RS} = \texttt{0x40}|\texttt{0x01} = \texttt{0x41} \tag{3}$$

$$\text{Low Nibble} = ((\texttt{0x48} << 4)\&\texttt{0xF0})|\text{RS} = \texttt{0x80}|\texttt{0x01} = \texttt{0x81} \tag{4}$$

With Enable and Backlight:

$$\text{High+EN+BL} = \texttt{0x41}|\texttt{0x04}|\texttt{0x08} = \texttt{0x4D} \tag{5}$$

$$\text{High+BL} = \texttt{0x41}|\texttt{0x08} = \texttt{0x49} \tag{6}$$

$$\text{Low+EN+BL} = \texttt{0x81}|\texttt{0x04}|\texttt{0x08} = \texttt{0x8D} \tag{7}$$

$$\text{Low+BL} = \texttt{0x81}|\texttt{0x08} = \texttt{0x89} \tag{8}$$

5.3 Complete Signal Timing for "Hello VLSI"

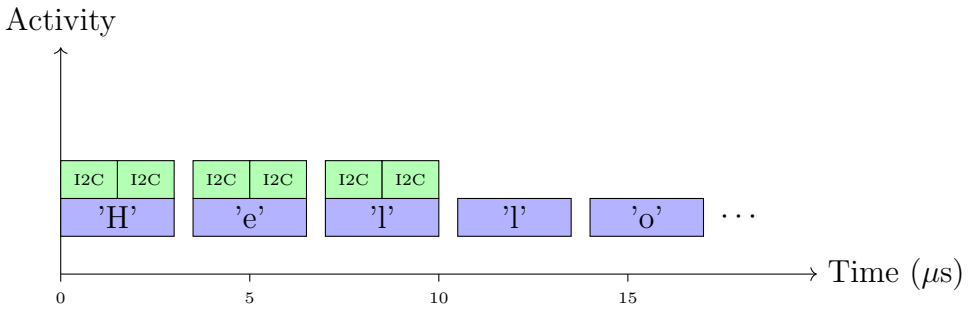


Figure 12: Timeline for "Hello VLSI" String Transmission

6 VLSI Concepts Demonstrated

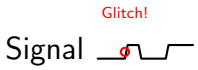
6.1 Atomicity and Race Conditions

Problem: If pin changes require multiple CPU instructions, interrupt handlers or concurrent processes could read intermediate states.

Solution: The BSRR register guarantees atomic read-modify-write operations.

Non-Atomic (WRONG):
1. Read ODR
2. Modify bit
3. Write ODR

→ 3 clock cycles



Atomic (CORRECT):
1. Write to BSRR

→ 1 clock cycle

Figure 13: Atomic vs Non-Atomic Pin Operations

6.2 Setup and Hold Time Requirements

Digital circuits require data to be stable before and after the clock edge.

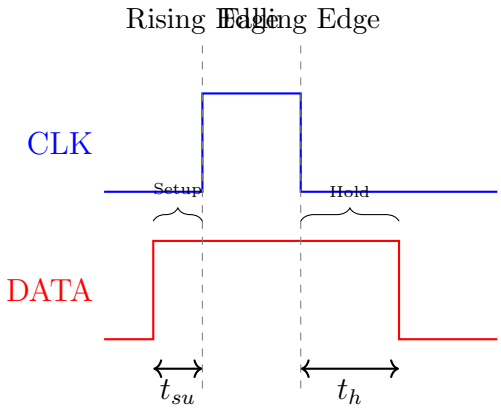


Figure 14: Setup and Hold Time Illustration

Timing Violations

- **Setup Violation:** Data changes too close to clock edge → Metastability
- **Hold Violation:** Data changes too soon after clock edge → Wrong data latched

Both violations can cause unpredictable behavior!

6.3 Bus Arbitration and Electrical Contention

Why Open-Drain?

If two devices drive the same line in opposite directions (one HIGH, one LOW), a **short circuit** occurs:

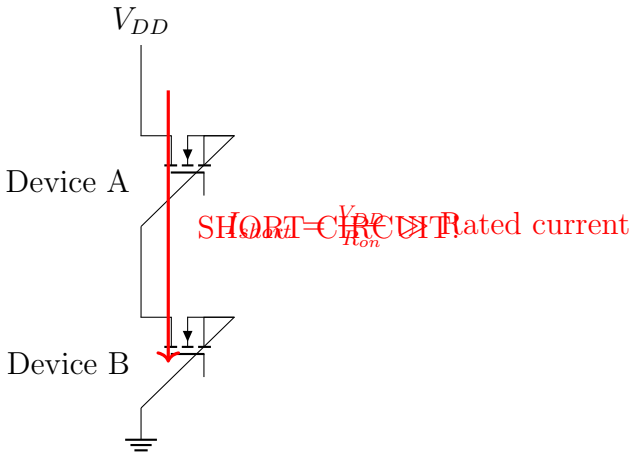


Figure 15: Bus Contention in Push-Pull Configuration

Open-Drain Solution: Wired-AND logic prevents conflicts

Device A	Device B	Bus State
OFF (Hi-Z)	OFF (Hi-Z)	HIGH (pull-up)
OFF (Hi-Z)	ON (Pull LOW)	LOW
ON (Pull LOW)	OFF (Hi-Z)	LOW
ON (Pull LOW)	ON (Pull LOW)	LOW

Table 3: Open-Drain Wired-AND Truth Table

7 LCD Initialization Sequence

The LCD requires a specific initialization sequence to enter 4-bit mode:

```
void LCD_Init(void) {
    HAL_Delay(50);            // Wait for LCD power stabilization

    LCD_Send(0x33, 0);        // Initialize: 8-bit mode
    HAL_Delay(5);

    LCD_Send(0x32, 0);        // Switch to 4-bit mode

    LCD_Send(0x28, 0);        // Function Set:
                              // - 4-bit interface
                              // - 2-line display
                              // - 5x8 dot character font

    LCD_Send(0x0C, 0);        // Display Control:
                              // - Display ON
                              // - Cursor OFF
                              // - Blink OFF

    LCD_Send(0x01, 0);        // Clear Display
    HAL_Delay(2);             // Clear requires ~1.5ms
}
```

Listing 7: LCD Initialization

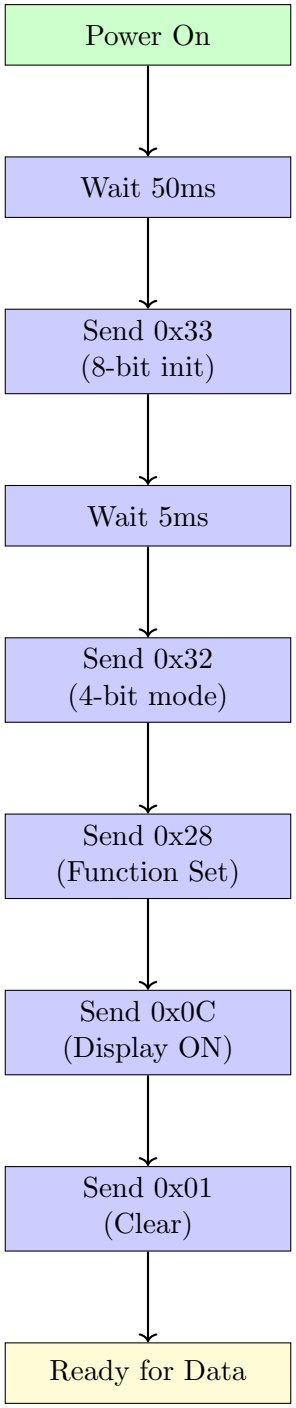


Figure 16: LCD Initialization Flowchart

8 Performance Analysis

8.1 Timing Calculations

For the I2C delay function:

```
void I2C_Delay(void) {
    for(volatile int i = 0; i < 50; i++);
}
```

Assuming each iteration takes approximately 3 CPU cycles at 16 MHz:

$$t_{delay} = \frac{50 \times 3}{16 \times 10^6} = 9.375\,\mu\text{s} \tag{9}$$

For one I2C byte transmission (8 data bits + 1 ACK):

$$\begin{aligned} t_{byte} &= 9 \times (2 \times t_{delay} + t_{SCL\,high} + t_{SCL\,low}) && (10)\\ &\approx 9 \times (2 \times 9.375 + 9.375 + 9.375)\,\mu\text{s} && (11)\\ &= 253.125\,\mu\text{s} && (12) \end{aligned}$$

Effective I2C frequency:

$$f_{I2C} = \frac{1}{t_{byte}/8} = \frac{8}{253.125 \times 10^{-6}} \approx 31.6\,\text{kHz} \tag{13}$$

8.2 Character Transmission Time

For each character, we send:

- 4 I2C transactions (high nibble E=1, E=0, low nibble E=1, E=0)
- Each transaction: 1 address byte + 1 data byte

$$\begin{aligned} t_{char} &= 4 \times 2 \times t_{byte} && (14)\\ &= 8 \times 253.125\,\mu\text{s} && (15)\\ &= 2.025\,\text{ms per character} && (16) \end{aligned}$$

For "Hello VLSI" (10 characters):

$$t_{total} = 10 \times 2.025\,\text{ms} = 20.25\,\text{ms} \tag{17}$$

9 Code Organization and Architecture

9.1 Layered Software Architecture

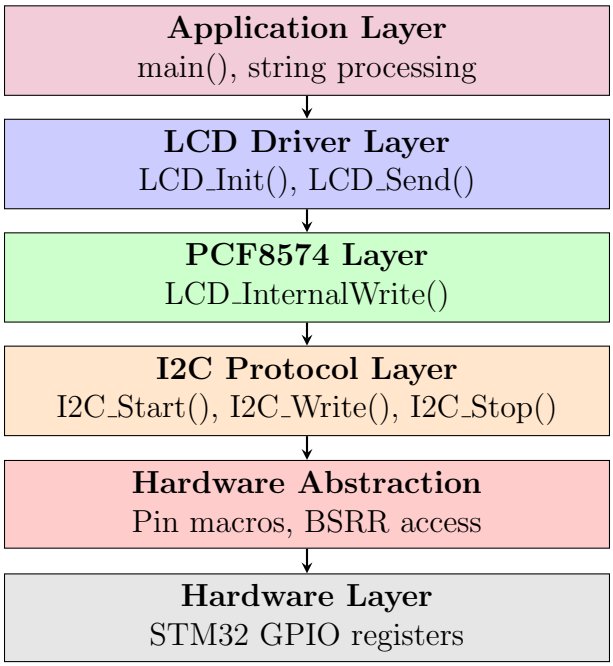


Figure 17: Software Layer Abstraction

9.2 Function Call Hierarchy

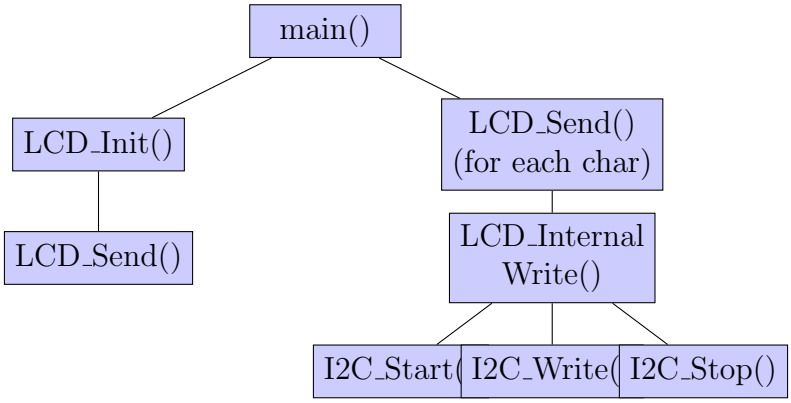


Figure 18: Function Call Tree

10 Testing and Debugging Strategies

10.1 Signal Integrity Verification

To verify correct I2C timing, use an oscilloscope or logic analyzer:

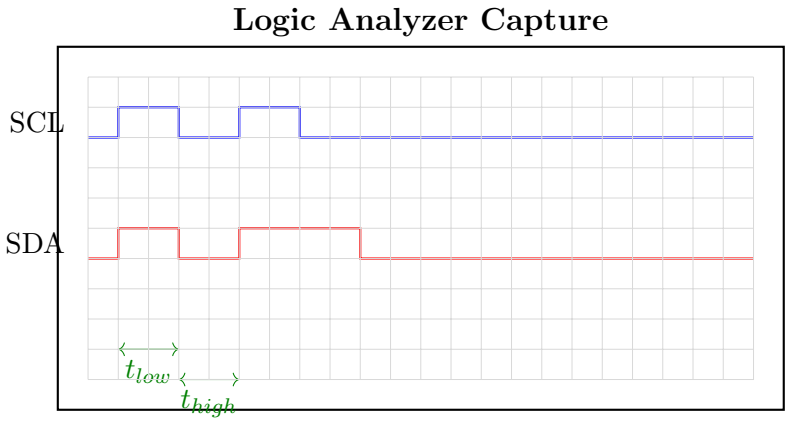


Figure 19: Expected Logic Analyzer Output

10.2 Common Issues and Solutions

Symptom	Likely Cause	Solution
No display output	Initialization failed	Increase delays in LCD_Init()
Garbled characters	Timing violation	Increase I2C_Delay()
Missing ACK	Wrong I2C address	Verify LCD_ADDR value
Random glitches	Non-atomic operations	Use BSRR instead of ODR
Backlight off	BL_BIT not set	Verify bit OR operations

Table 4: Debugging Guide

11 Extensions and Improvements

11.1 Potential Enhancements

1. **DMA-Based I2C:** Use hardware I2C peripheral with DMA for higher performance
2. **Interrupt-Driven Timing:** Replace busy-wait delays with timer interrupts
3. **Error Detection:** Check ACK bits and implement retry logic
4. **Multiple Devices:** Extend to support multiple I2C slaves
5. **Clock Stretching:** Handle slaves that hold SCL low

11.2 Performance Optimization

Current implementation vs. optimized:

Method	Speed	CPU Usage
Current (Bit-banging)	31.6 kHz	100% (blocking)
Hardware I2C	100-400 kHz	¡5% (with DMA)
Hardware I2C + DMA	400 kHz	¡1% (background)

Table 5: Performance Comparison

12 Conclusion

12.1 Key Achievements

This project successfully demonstrates:

- **Hardware Control:** Direct manipulation of GPIO registers for protocol implementation
- **Protocol Mastery:** Manual implementation of I2C timing and sequencing
- **Interface Design:** Custom driver for PCF8574 I/O expander
- **System Integration:** Complete working system from CPU to LCD pixels
- **VLSI Concepts:** Practical application of timing analysis, atomicity, and bus arbitration

12.2 Learning Outcomes

VLSI Skills Demonstrated

1. Understanding of digital timing requirements (setup/hold times)
2. Knowledge of open-drain vs. push-pull output configurations
3. Experience with serial communication protocols
4. Ability to read hardware datasheets and implement specifications
5. Debugging skills for hardware-software interface issues
6. Understanding of the gap between high-level software and silicon behavior

12.3 Portfolio Value

For VLSI engineering positions, this project proves:

- You don't just use libraries—you *understand* what they do
- You can work at any level of abstraction (application to gates)
- You understand electrical constraints, not just logical behavior

- You can implement industry-standard protocols from scratch

"Understanding silicon means understanding the dance between voltage, time, and logic—this project orchestrates that dance."

References

1. STM32G0 Reference Manual, STMicroelectronics
2. I2C-bus specification and user manual, NXS Semiconductors, Rev. 7.0, 2021
3. PCF8574/PCF8574A Datasheet, Texas Instruments
4. HD44780U LCD Controller Datasheet, Hitachi
5. ARM Cortex-M0+ Technical Reference Manual