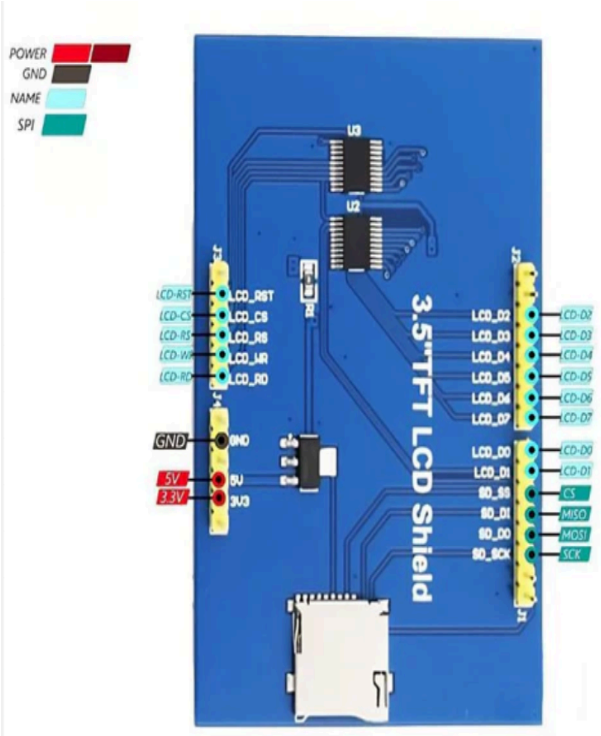
software

Esp32 and 3.5”TFT LCD Shield Project Report



3.5” TFT LCD shield Display **ESP32**

INDEX

DESCRIPTION

ESP32

The ESP32 dev module is a microcontroller development board featuring a dual-core processor with integrated Wi-Fi and Bluetooth, making it ideal for IoT and embedded projects. It typically includes a micro-USB for power and programming, various peripherals like ADC, DAC, I2C, and UART, and is designed for low-power applications. The board's design breaks out many of the main chip's GPIO pins for easy connection to sensors and other components

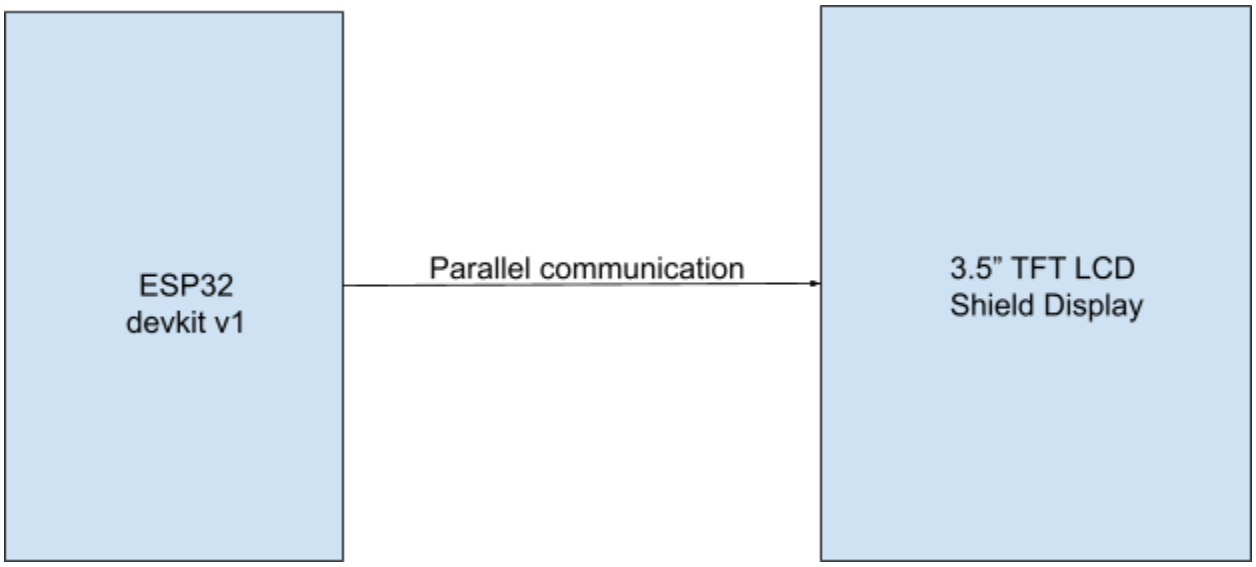
- **Processing**: Dual-core LX6 microprocessor running up to 240MHz
- **Connectivity**: Integrated 2.4 GHz Wi-Fi and Bluetooth
- **Memory**: Typically includes up to 512 KB RAM and 4 MB or more of flash memory.
- **Peripherals**: Includes a wide range of peripherals such as ADC, DAC, I2C, SPI, I2S, UART, and PWM, along with capacitive touch sensors
- **User interface**: Features a USB-to-UART bridge, a built-in LED, reset and boot buttons, and is designed for easy access to most GPIO pins.

3.5”TFT LCD Shield

A 3.5" TFT LCD shield is a plug-and-play display module for microcontrollers, features a 3.5-inch color screen with 480x320 resolution, four white LED backlights, and individual pixel control.It connects to the microcontroller via different modes (SPI- 4line,and 3 line, 16 bit mode, 9 bit mode, 8 bit mode) interface and often includes an on-board MicroSD card slot for data storage.

- **Display**: 3.5-inch diagonal TFT LCD with 18-bit color, capable of displaying 262,000 shades.
- **Resolution**: 480x320 pixels for detailed and sharp images.
- **Backlight**: Four white LEDs for bright, clear illumination.
- **MicroSD card slot**: An onboard slot provides easy storage for images, fonts, and other data.
- **Controller**: Common controllers include the ILI9486, which allows for drawing graphics, text, and pictures.
- **Compatibility**: Designed to be compatible with a wide range of development boards, including Arduino Uno, Leonardo, and others.
- **Ease of use**: Comes fully assembled and requires no soldering; it can be used by simply plugging it in and loading the appropriate library.

Block Diagram:



- Here’s a **clear block diagram explanation** and description for connecting an **ESP32** to a **3.5" TFT LCD Shield display** (usually an ILI9488-based 480x320 screen)
- Common Setup: ESP32 + 3.5" TFT LCD parallel connections which use **8-bit parallel buses** (D0–D7 + control pins).
- We can adapt these to the **ESP32** using libraries like LCD_DRIVER.h” defining the libraries in /LCD_DRIVER.cpp both of which support **parallel displays**.

Pin Configurations:

TFT LCD Display GPIO pin	ESP GPIO pin
LCD_D0	D12
LCD_D1	D13
LCD_D2	D26
LCD_D3	D25
LCD_D4	D33
LCD_D5	D32
LCD_D6	D27
LCD_D7	D14
LCD_RD	D19
LCD_WR	D2
LCD_RS	D4
LCD_CS	D15
LCD_RST	D5

Functional Summary

- **D0–D7** → 8-bit **parallel data bus**.
  The ESP32 sends 8 bits of data (one byte) at a time to the display. This data can represent either a **command** or **pixel color data**, depending on the **RS (Register Select)** signal.

- **WR (Write Strobe)** → Controls when data is latched into the LCD.
  The LCD reads the value on D0–D7 on the **falling edge** of WR.

- **RS (Register Select)** → Chooses what kind of information is being sent:

  - **LOW** = Command (like "Set Column Address")

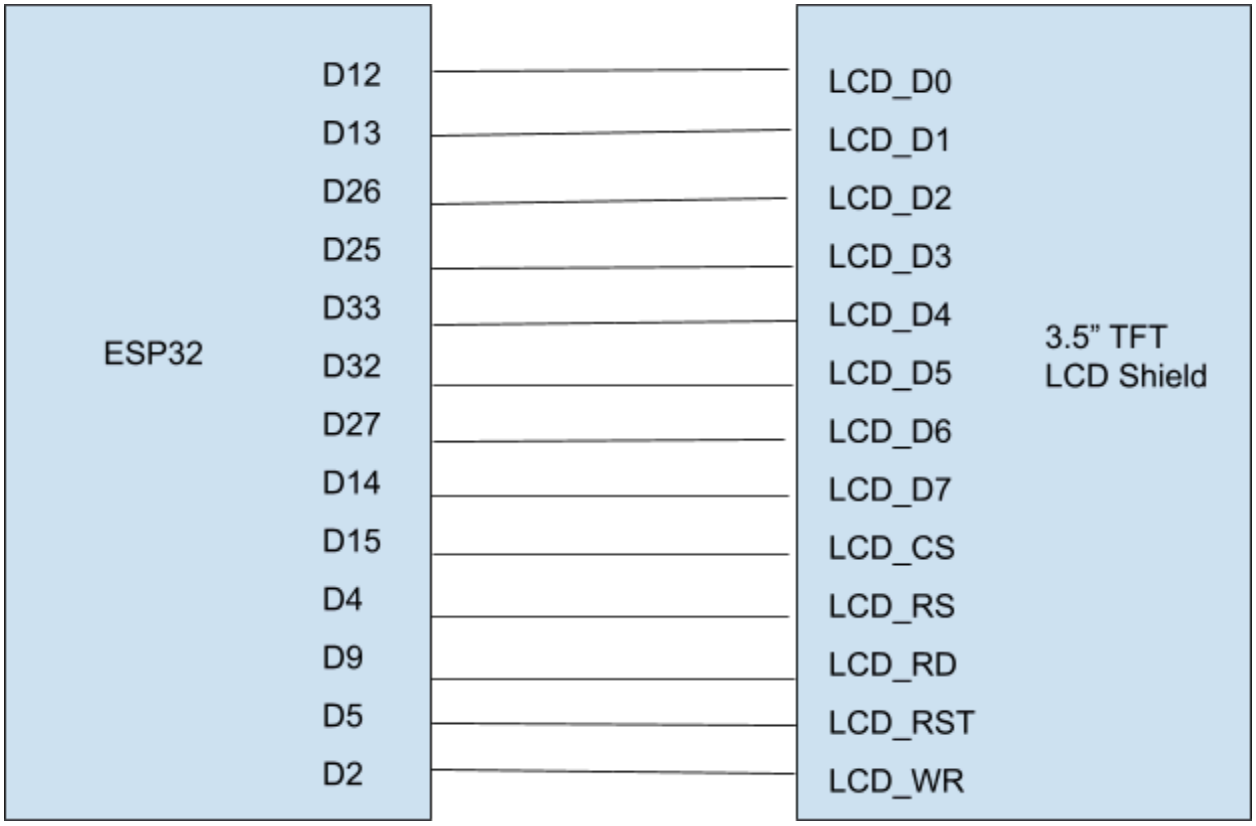
  - **HIGH** = Data (like pixel color value)

- **CS (Chip Select)** → Enables the LCD.
  When multiple SPI/parallel devices share the bus, this ensures only one is active at a time.

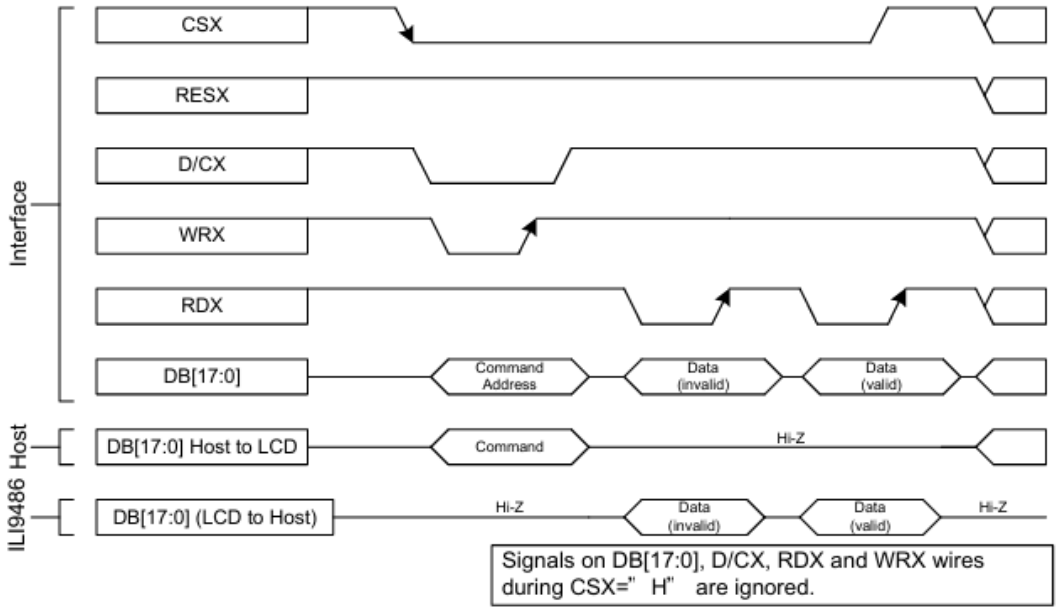
- **RD (Read Strobe)** → Optional. Used to read LCD's internal data (rarely needed unless touch or status check). Often tied HIGH.

- **RST (Reset)** → Ensures the LCD starts from a known state after power-up.

Wiring diagram:



Timing sequence:-



1 CSX (Chip Select)

- **Active LOW signal.**
- When **CSX = LOW**, the LCD is selected and will accept data or commands.
- When **CSX = HIGH**, all bus activity is **ignored** (as noted in the diagram).

In the timing chart:
At the left edge, **CSX goes LOW**, enabling the LCD interface.

---

2 D/CX (Data/Command Select)

- Determines whether the next byte is a **command** or **data**.
    - **LOW** → Command
    - **HIGH** → Data

Sequence:

- First, **D/CX = LOW** to send a **command**.
- Later, it switches **HIGH** to send **data** related to that command.

---

3 WRX (Write Strobe)

- Each **falling edge** of WRX latches one word (8, 16, or 18 bits) from the data bus (**DB[17:0]**) into the LCD controller.

Sequence:

- When **WRX** goes LOW → data on DB[17:0] is **latched** into LCD.
- When **WRX** returns HIGH → ready for next data/command.

---

4 RDX (Read Strobe)

- Used when the host wants to **read** data (e.g., display status or pixel color).
- When **RDX = LOW**, LCD places data on DB[17:0].
- Otherwise, the data bus is **Hi-Z (high impedance)** — meaning it's disconnected to prevent conflicts.

Sequence:

- During write, **RDX stays HIGH** (inactive).

- During read, it toggles LOW → HIGH to capture data.

---

5 DB[17:0] (Data Bus)

- Represents the parallel data lines.
- When sending:
    - **Command phase:** D/CX = LOW → DB carries command code.
    - **Data phase:** D/CX = HIGH → DB carries pixel/color data.

Sequence:

1. **Command Address Phase —** MCU sends command (e.g., “Write Memory Start”).
2. **Data (invalid) —** brief settling period.
3. **Data (valid) —** actual pixel or register data sent.

---

6 Direction Control

There are two modes:

- **Host → LCD:** During write operations (commands/data).
- **LCD → Host:** During read operations (e.g., reading display status).

In the diagram:

- “**DB[17:0] Host to LCD**” → during command and data write.
- “**DB[17:0] LCD to Host**” → during data readback (e.g., pixel data).
- **Hi-Z** means the bus is not driven by either device.

---

7 Key Rule

"Signals on DB[17:0], D/CX, RDX, and WRX wires during CSX = HIGH are ignored."

Drivers debug:

File Name: LCD_DRIVER.cpp

```
#include "LCD_DRIVER.h"

#include <arduino.h>


CLCD_Driver::CLCD_Driver()
{
	m_sLcd.pin_num_cs = 15;

	m_sLcd.pin_num_wr = 2; //LCD_WR

	m_sLcd.pin_num_rd = 19; //LCD_RD

	m_sLcd.pin_num_rs = 4;

	m_sLcd.pin_num_rst = 5;

	m_sLcd.pin_data_num[0] = 12;//LCD_D0

	m_sLcd.pin_data_num[1] = 13;

	m_sLcd.pin_data_num[2] = 26;

	m_sLcd.pin_data_num[3] = 25;

	m_sLcd.pin_data_num[4] = 33;

	m_sLcd.pin_data_num[5] = 32;

	m_sLcd.pin_data_num[6] = 27;

	m_sLcd.pin_data_num[7] = 14;

}

void CLCD_Driver::WriteCommand(unsigned char yData)
{

}


void CLCD_Driver::WriteData(unsigned char yData)
{

	digitalWrite(m_sLcd.pin_num_cs, 0);
	digitalWrite(m_sLcd.pin_num_rs,1); //enabling the data mode

	digitalWrite(m_sLcd.pin_num_wr,1); //enable write pin
```

```
	for(int  i = 0; i < 8; i++)
	{
		if( yData & 0x80)
		{
			digitalWrite(m_sLcd.pin_data_num[7-i], 1);
		}
		else
		{
			digitalWrite(m_sLcd.pin_data_num[7-i], 0);
		}

		yData <<= 1;
	}


	digitalWrite(m_sLcd.pin_num_wr,0); //disable write pin

}


void CLCD_Driver::InitPins_mode()
{
   char ayMsg[512];//char data
	pinMode(m_sLcd.pin_num_wr, OUTPUT); //write pin as op(set to 0)
	//digitalWrite(m_sLcd.pin_num_wr,0);

	pinMode(m_sLcd.pin_num_cs, OUTPUT); //cs as op(set to 0)
	digitalWrite(m_sLcd.pin_num_cs,0);

	pinMode(m_sLcd.pin_num_rd, OUTPUT); //read pin as output(set to 1)
	//digitalWrite(m_sLcd.pin_num_rd,1);

	pinMode(m_sLcd.pin_num_rs, OUTPUT); //register select to command or write mode

	pinMode(m_sLcd.pin_num_rst, OUTPUT); //reset pin

	pinMode(m_sLcd.pin_data_num[0], OUTPUT); //datapins 0-7

	pinMode(m_sLcd.pin_data_num[1], OUTPUT);

	pinMode(m_sLcd.pin_data_num[2], OUTPUT);

	pinMode(m_sLcd.pin_data_num[3], OUTPUT);

	pinMode(m_sLcd.pin_data_num[4], OUTPUT);

	pinMode(m_sLcd.pin_data_num[5], OUTPUT);

	pinMode(m_sLcd.pin_data_num[6], OUTPUT);

	pinMode(m_sLcd.pin_data_num[7], OUTPUT);

	// serialflash();


	sprintf(ayMsg,"d0-D7 = 0x55\r\n Press Enter to continue\r\n");
```

```
        Serial.println(ayMsg);
        WriteData(0x55);
        do
        {
                delay(1000);
        }while(!Serial.available());
        serialflash();

        sprintf(ayMsg,"d0-D7 = 0x55\r\n Press Enter to continue\r\n");
        Serial.println(ayMsg);
        LCD_data(0x55);
        do
        {
                delay(1000);
        }while(!Serial.available());
        serialflash();

        sprintf(ayMsg,"d0-D7 = 0xAA\r\n Press Enter to continue\r\n");
        Serial.println(ayMsg);
        WriteData(0xAA);
        do
        {
                delay(1000);
        }while(!Serial.available());
        serialflash();

        sprintf(ayMsg,"d0-D7 = 0xAA\r\n Press Enter to continue\r\n");
        Serial.println(ayMsg);
        LCD_data(0xAA);
        do
        {
                delay(1000);
        }while(!Serial.available());
        serialflash();
//      int yData = 0x55;

//      for(int  i = 0; i < 8; i++)

//      {
//              if((yData>>i) & 0x01)
//              {
//                      digitalWrite(m_sLcd.pin_data_num[i], 1);
//              }
//              else
//              {
//                      digitalWrite(m_sLcd.pin_data_num[i], 0);
//              }
//      }
}

void CLCD_Driver::serialflash()
{
        while(Serial.available())
        {
                char t = Serial.read();
        }
```

```
}

void CLCD_Driver::LCD_cmd(uint8_t command)
{

        digitalWrite(m_sLcd.pin_num_cs, 0);    // select LCD

        digitalWrite(m_sLcd.pin_num_rs,0); //enabling the command mode

        //digitalWrite(m_sLcd.pin_num_wr,1); //enable write pin

        Serial.print("Sending Command: 0x");
        Serial.println(command, HEX);

        for(int  i = 0; i < 8; i++)
        {
                if((command>>i) & 0x01)
                {
                        digitalWrite(m_sLcd.pin_data_num[i], 1);
                }
                else
                {
                        digitalWrite(m_sLcd.pin_data_num[i], 0);
                }
        }
        digitalWrite(m_sLcd.pin_num_wr, 0);
   delayMicroseconds(1);  // small delay
   digitalWrite(m_sLcd.pin_num_wr, 1);

   digitalWrite(m_sLcd.pin_num_cs, 1);   // deselect
}
void CLCD_Driver::LCD_data(uint8_t data){
        digitalWrite(m_sLcd.pin_num_cs, 0);
        digitalWrite(m_sLcd.pin_num_rs,1); //enabling the data mode
        digitalWrite(m_sLcd.pin_num_wr,1); //enable write pin

        for(int  i = 0; i < 8; i++)
        {
                if((data>>i) & 0x01)
                {
                        digitalWrite(m_sLcd.pin_data_num[i], 1);
                }
                else
                {
                        digitalWrite(m_sLcd.pin_data_num[i], 0);
                }
        }
        digitalWrite(m_sLcd.pin_num_wr,0); //disable write pin

}
CLCD_Driver::~CLCD_Driver()
{

}
```

File Name: LCD_DRIVER.h

```
#ifndef LCD_DRIVER_H
#define LCD_DRIVER_H

#include <stdint.h>

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;

// #include <ardunio.h>
/**
 * @brief Configuration of i2s lcd mode
 */
typedef struct {
    uint8_t data_width;          /*!< Parallel data width, 16bit or 8bit available */
    uint8_t pin_data_num[16];     /*!< Parallel data output IO*/
    uint8_t pin_num_wr;          /*!< Write clk io*/
    uint8_t pin_num_rd;
    uint8_t pin_num_cs;          /*!< CS io num */
    uint8_t pin_num_rs;          /*!< RS io num */
          uint8_t pin_num_rst;
          int clk_freq;               /*!< I2s clock frequency */
    //i2s_port_t i2s_port;        /*!< I2S port number */
    bool swap_data;             /*!< Swap the 2 bytes of RGB565 color */
    uint32_t buffer_size;       /*!< DMA buffer size */
} i2s_lcd_config_t;

class CLCD_Driver
{

void serialflash();
private:

  i2s_lcd_config_t  m_sLcd;

// The class

public:           // Access specifier

          void WriteCommand(unsigned char yData);

          void WriteData(unsigned char yData);

          CLCD_Driver();
          ~CLCD_Driver();

          void InitPins_mode();
          void LCD_cmd(uint8_t command);
          void LCD_data(uint8_t data);
};
```

```
#endif  //LCD_DRIVER_H
```

Hardware

Hardware Design:

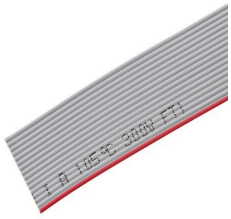
1. Introduction

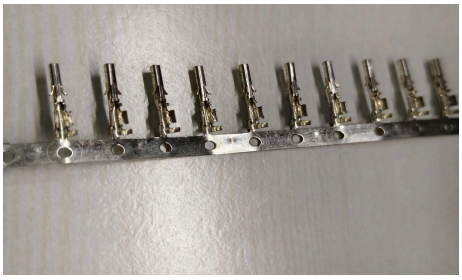
- This project demonstrates how to interface a **3.5-inch TFT LCD Touch Display** with an **ESP32 controller** using **parallel communication**.
- The purpose is to display graphical content such as text, images, and UI elements on the LCD screen using the ESP32 microcontroller.
- This setup helps in understanding **GPIO-based parallel data transfer**, **display drivers**, and **touch interface concepts**.

---

2. Components Required

S. No	Component	Description
1.	ESP32 Development Board	Main microcontroller with dual-core CPU, Wi-Fi, Bluetooth, and multiple GPIOs.
2.	3.5" TFT LCD Shield	Touchscreen display module that communicates through an 8-bit or 16-bit parallel data bus.
3	FRC Cable	Used to make connections between ESP32 and LCD.
4	USB Cable	To power and program the ESP32
5	Laptop/PC	For uploading code using Arduino IDE.





3. Hardware Connections

The parallel interface uses multiple data lines (D0–D7 or D0–D15) and control pins like RS, WR, CS, and RST.

Each bit of data is sent simultaneously through these pins.

3. Pin Configuration

The TFT LCD communicates with the ESP32 using an **8-bit parallel data bus** and several **control pins** for command, read/write, and reset functions.

TFT LCD Display GPIO Pin	ESP32 GPIO Pin	Description
LCD_D0	D12	Data bit 0
LCD_D1	D13	Data bit 1
LCD_D2	D26	Data bit 2
LCD_D3	D25	Data bit 3
LCD_D4	D33	Data bit 4
LCD_D5	D32	Data bit 5
LCD_D6	D27	Data bit 6
LCD_D7	D14	Data bit 7
LCD_RD	D19	Read signal
LCD_WR	D2	Write signal
LCD_RS	D4	Register select (Command/Data)

LCD_CS	D15	Chip select
LCD_RST	D5	Reset pin
VCC	3.3V	Power supply
GND	GND	Ground

4. Software Requirements

- Arduino IDE
- ESP32 board package installed

5. Working Principle

1. The ESP32 sends pixel or command data to the LCD using an 8-bit parallel bus.
2. Each data byte is written when the WR pin is pulsed low.
3. Control pins (RS, CS, RST) determine whether the data is a command or display content.
4. The LCD displays the received data immediately on screen.
5. If touch functionality is available, additional pins (like T_CLK, T_CS, T_DIN, T_DOUT, T_IRQ) can be connected to read user touch inputs via SPI or analog pins.

Steps to Connect LCD Display to ESP32 using Ribbon Cable:

1. Take one ribbon cable and cut the edge neatly.

2. Take one crimping pin and insert one wire from the ribbon cable into the crimping pin.

3. Use a crimping tool to press (tighten) the pin properly so that the wire is fixed strongly.

4. Repeat this process for all 15 GPIO wires needed for the LCD connection.

5. Once all wires are crimped, connect them properly using a connector — match each LCD pin to the correct ESP32 GPIO pin.

6. After completing the connections, use a multimeter to check continuity and ensure all connections are fine.

7. If all connections are correct and working properly, then proceed to the software part for display testing.

Tab 3