Fayoum University

Engineering Faculty

Electrical Engineering Department

B. Eng. Final Year Project

Secure Communication Network

By:

Ahmed Abdulla Mourad

Ahmed Eid Mohamed

Mohamed Shaban El-sayed

Moaz Mohamed Radwan

Supervised By:

Dr. Mohamed Hamdy

Dr: Mohamed Hamdy *1, July 2025*

July – 2025

DEDICATION

We would like to dedicate this project to all those who supported and guided us throughout this journey:
To Dr. Mohamed Hamdy,
For your invaluable guidance in choosing the project idea and your constant support in developing and refining it. Your mentorship was a key pillar in turning this vision into reality.
To the college management,
For your dedication to fostering a strong academic environment, and for providing us with the tools, laboratories, and resources that made this work possible.
With sincere gratitude, we dedicate this work to all of you.
To our beloved families,
For your unwavering love, encouragement, and sacrifices. Your support gave us strength and motivation every step of the way. This achievement belongs to you as much as it does to us.

ACKNOWLEDGMENT

invaluable supervision, continuous guidance, and encouragement throughout every stage of this project. His insight and experience played a vital role in shaping the idea and bringing it to life.

We would also like to extend our sincere appreciation to the faculty members of the Electronics and Communication Engineering Department. Their dedication, support, and the knowledge they shared with us have been instrumental in building the foundation of this work.

Our heartfelt thanks go to the college management and technical staff for providing us with access to laboratories, tools, and essential components. Their support and cooperation enabled us to carry out practical experiments and implement our project effectively.

We gratefully acknowledge the Scientific Research Organization for their financial support, which was crucial in enabling us to acquire the necessary resources and tools for this project.

Finally, we are deeply thankful to our families for their patience, encouragement, and emotional support throughout our academic journey. Their belief in us gave us strength and motivation during every step of this work.

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Bachelor of Science in *Electrical Engineering* is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____
Registration No.: _____
Date: Day, xx Month Year.

ABSTRACT

This project presents the development of a secure networking terminal designed for offline message and audio communication using a Raspberry Pi and a TFT touchscreen interface. The system is tailored for environments where internet access is unavailable or unreliable, such as remote field operations, military zones, or disaster response areas. It allows users to easily send and receive text and voice messages through a graphical user interface (GUI) powered by the LVGL graphics library, ensuring a user-friendly interaction experience.

The core hardware includes a Raspberry Pi, a TFT touch display (ILI9341), a touch controller (XPT2046), a transceiver module with up to 500 meters of wireless range, a microphone, and a speaker. Software communication with the display and touch input is handled via the BCM2835 SPI library. The system facilitates message encryption, user interaction through the touch interface, voice recording, playback, and real-time message exchange over the transceiver. This report details the system architecture, hardware integration, software design, user interface implementation, and testing results. The project demonstrates the feasibility of developing a reliable and portable secure communication device without dependence on internet infrastructure.

TABLE OF CONTENTS

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE SNIPPETS

LIST OF ACRONYMS/ABBREVIATIONS

ACRONYM	Definition of Acronym
ACK	Acknowledge
ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
ALSA	Advanced Linux Sound Architecture
APT	Advanced Package Tool
ARM	Advanced RISC Machine
ARP	Address Resolution Protocol
CABC	Content Adaptive Brightness Control
CBC	Cipher Block Chaining
CIA Triad	Confidentiality, Integrity, and Availability Triad
CPU	Central Processing Unit
CSN	Chip Select Not
DDRAM	Display Data RAM
DE	Data Enable
DIM	Brightness transition time
DNS	Domain Name System
DoS	Denial-of-Service
DPI	Dots Per Inch
ECU	Electronic Control Unit
FIN	Finish Packet
GDPR	General Data Protection Regulation
GPIO	General-Purpose Input/Output
GPU	Graphics Processing Unit
GRAM	Graphics RAM
GUI	Graphical User Interface
HBP	Horizontal Back Porch
HFP	Horizontal Front Porch
HDMI	High-Definition Multimedia Interface
HSYNC	Horizontal Synchronization
IC	Integrated Circuit

IDE	Integrated Development Environment
ILI9341	Display Driver IC
IoT	Internet of Things
IP	Internet Protocol
IRQ	Interrupt Request
ISM	Industrial, Scientific, and Medical radio band
ISO	International Organization for Standardization
IV	Initialization Vector
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
LSB	Least Significant Bit
LUT	Look-Up Table
LVGL	Light and Versatile Graphics Library
LXDE	Lightweight X11 Desktop Environment
MADCTL	Memory Access Control
MCU	Microcontroller Unit
MISO	Master In Slave Out
MITM	Man-in-the-Middle
MOSI	Master Out Slave In
MP3	MPEG-1 Audio Layer 3
MSB	Most Significant Bit
MVP	Minimum Viable Prototype
NIST	National Institute of Standards and Technology
NOP	No-Operation
OS	Operating System
P2P	Peer-to-Peer
PA	Power Amplifier
PMP	Portable Multimedia Players
PWM	Pulse Width Modulation
RAM	Random Access Memory
RCA	Radio Corporation of America
RESX	Reset
RF	Radio Frequency
RGB	Red Green Blue

RIM	RGB interface mode
RPI	Raspberry Pi
RSA	Rivest-Shamir-Adleman
SCK	Serial Clock
SCL	Serial Clock Line
SDA	Serial Data Input/Output
SDI	Serial Data Input
SDO	Serial Data Output
SHA	Secure Hash Algorithm
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SYN	Synchronize
SYN-ACK	Synchronize-Acknowledge
TCP	Transmission Control Protocol
TE	Tearing Effect
TFT	Thin-Film Transistor
TH	Threshold
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
UI	User Interface
ULP	Ultra-Low Power
USB	Universal Serial Bus
VBP	Vertical Back Porch
VCI	Analog supply voltage
VCOM	Common Electrode Voltage
VFP	Vertical Front Porch
VoIP	Voice over Internet Protocol
Wi-Fi	Wireless Fidelity
WRX	Write Strobe
XPT2046	Touch Controller IC

Chapter One

1 INTRODUCTION

1.1 BACKGROUND

In today's interconnected world, communication has become the backbone of operations across all domains from civilian coordination to defense logistics. Yet, there are many scenarios where conventional communication infrastructure, such as mobile networks and internet connectivity, either becomes inaccessible or is deliberately disabled. Situations such as military deployments, natural disasters, border operations, and remote field activities often present significant communication challenges, particularly in terms of maintaining operational confidentiality, availability, and independence from third-party networks. These limitations pose serious risks not only to mission effectiveness but also to safety and coordination on the ground.

The growing reliance on internet-based communication platforms has further increased vulnerability in high-stakes environments, where cyber threats, surveillance, or service disruption can compromise both the confidentiality and reliability of communication. Recognizing these concerns, various countries have begun to develop and adopt offline communication solutions tailored to national needs. For instance, recent developments in Lebanon have seen the deployment of pager-based systems that allow for decentralized, offline communication during critical events. These initiatives reflect a global awareness of the necessity for sovereign communication technologies that do not rely on commercial or external infrastructure.

Despite the strategic importance of such systems, Egypt currently lacks a secure, locally developed communication solution that can operate autonomously in offline conditions. Existing alternatives tend to be imported, limited in adaptability, or not designed with local operational contexts in mind. This gap highlights a national need for an innovative, self-contained system that ensures secure and reliable communication independent of foreign technologies and infrastructure.

In response to this need, this project proposes the development of a Secure Networking Terminal a platform that supports offline communication through secure, private wireless channels. The system is envisioned as a strategic tool for enhancing local capabilities in sensitive and resource-constrained environments. It represents not only a technical challenge but also an opportunity to contribute to technological sovereignty by developing core communication infrastructure.

1.2 PROBLEM STATEMENT

Lack of Offline Communication Solutions
Organizations—including companies, emergency teams, and field operations—often operate in areas with unreliable internet or cellular coverage. National efforts toward digital connectivity remain uneven, with rural and infrastructure-challenged regions particularly affected. This gap leaves critical operations vulnerable when existing networks fail.

Dependence on Foreign Technologies
Egypt heavily relies on imported communication hardware and proprietary software, reinforcing vendor lock-in and reducing control over system customization. This dependence limits local innovation and autonomy in communications technology and drives higher costs.

Centralized Infrastructure & Network Control
Telecom infrastructures are centrally managed by a few major players—Telecom Egypt, Vodafone, Orange—granting the government legal powers to shut down or throttle services during emergencies or security events. This introduces a single point of failure for conventional communications.

Insufficient Security in Existing Tools
Popular corporate communication channels typically rely on central servers and offer limited or absent end-to-end encryption in peer-to-peer, offline scenarios. This poses significant risks for organizations handling sensitive or confidential information.

Absence of Dual-Mode Offline Devices
While standalone offline systems exist—such as pagers or simple radios—they generally offer only text or voice channels, not both. Integrating voice and text into a single device remains rare, reducing user flexibility in practice.

Short Communication Range in Portable Devices
Common portable devices (e.g., walkie-talkies, Bluetooth tools) often struggle to support secure communication beyond tens of meters or require infrastructure boosters. A gap remains for affordable, standalone devices capable of secure messaging across distances up to 500 m.

1.3 SCOPE AND LIMITATIONS

Scope

This project focuses on the design and development of a minimum viable prototype (MVP) for a Secure Networking Terminal capable of offline peer-to-peer communication. The implemented system consists of two standalone nodes, each built using a Raspberry Pi 3. The devices feature an interactive touchscreen interface based on a TFT display, allowing users to send and receive both text and voice messages. Basic security is integrated through a custom encryption model involving RSA for session key exchange and AES for data confidentiality. The communication is established over a private wireless channel without reliance on the internet or mobile infrastructure. The system's graphical user interface is developed using embedded tools and libraries tailored for touch-based interaction.

Limitations

The current version of the project is a prototype and is subject to several limitations:

- **Limited Node Count**: The system supports only two devices in a one-to-one communication model. Group messaging or mesh networking is not included in the current scope.

- **Unverified Range**: The intended communication range is up to 500 meters; however, this has not yet been validated under real-world conditions and may vary depending on environmental and hardware factors.

- **Basic Encryption Model**: While the system implements RSA and AES, the encryption methods are not formally certified or subjected to third-party security audits.

- **No Internet or Network Fallback**: The system is designed for offline use only. It does not offer any integration with external networks or online communication platforms.

- **Hardware-Specific Implementation**: The current prototype is optimized for Raspberry Pi 3 hardware with a specific touchscreen model. Adaptability to other hardware platforms has not been tested.

- **User Interface Constraints**: The UI is basic and built for demonstration purposes. It lacks features such as contact management, delivery confirmation, or message history logging.

These limitations are acknowledged as part of the MVP stage and serve as a foundation for future enhancements in range, scalability, security, and usability.

1.4 METHODOLOGY OVERVIEW

This chapter outlines the methodology followed in the design, development, and testing of the Secure Networking Terminal. The project follows a modular and integration-driven development strategy, aiming to build a functional prototype that supports secure offline text and voice communication between two independent hardware nodes. The methodology encompasses both hardware and software aspects and includes the selection and configuration of electronic components, embedded programming in C, custom implementation of cryptographic routines, and user interface design. Each stage of the methodology is aligned with the goal of producing a minimum viable product that operates entirely offline, with no dependency on internet connectivity or external infrastructure.

1.4.1 Design Strategy

The development process was organized using a modular design approach. The project was decomposed into four primary modules, each representing a distinct functional unit: communication, encryption, audio processing, and graphical user interface (GUI). Each module was developed independently in both hardware and software layers, followed by a controlled integration phase.

- **Communication Module**: Built using NRF24L01 transceivers, this module enables wireless peer-to-peer message transmission between two Raspberry Pi 3 boards. It establishes the physical layer of the communication channel.

- **Encryption Module**: RSA was implemented for secure session key exchange, while AES was used for encrypting and decrypting message content. All cryptographic routines were written in C using the OpenSSL library, with manual integration into the messaging pipeline.

- **Voice Module**: Audio input and output were managed directly in C using ALSA-compatible functions. Voice messages could be recorded, encoded, transmitted, and replayed entirely offline using onboard microphone and speaker components.

- **User Interface Module**: The GUI was designed and developed using the LVGL graphics library, with interaction handled through an ILI9341 TFT touchscreen and an XPT2046 touch controller. The GUI enables users to compose, send, and receive both text and voice messages.

After individual development and unit testing of each module, a staged integration process was adopted:

1. First, the communication and encryption modules were merged to ensure secure message transmission.
2. Next, the GUI was combined with the voice module to support media interaction via touchscreen.
3. Finally, all modules were integrated into a unified system, with the Raspberry Pi managing coordination across components.

1.5 SYSTEM ARCHITECTURE

The Secure Networking Terminal was designed as a self-contained, embedded system integrating both hardware and software components to facilitate secure offline communication. The system architecture reflects a tightly coupled model where each hardware element directly supports a specific software functionality. The architecture follows a layered structure comprising user interaction, data processing, communication handling, and cryptographic protection.

At the core of each node is a **Raspberry Pi 3**, which functions as the central processing unit. It interfaces with the following components:

- **TFT Touchscreen (ILI9341)**: Displays the graphical user interface for composing and reading text messages, controlling voice recordings, and system feedback.

- **Touch Controller (XPT2046)**: Captures user input from the display and routes it to the UI logic.

- **Microphone and Speaker**: Handle voice recording and playback. These are directly interfaced via the Pi's audio I/O and managed through C routines.

- **NRF24L01 Transceiver Module**: Provides the wireless communication channel between nodes, operating in the 2.4 GHz ISM band with point-to-point configuration.

- **Local Storage**: Temporary buffers for storing voice messages and system logs (e.g., using the SD card or internal storage).

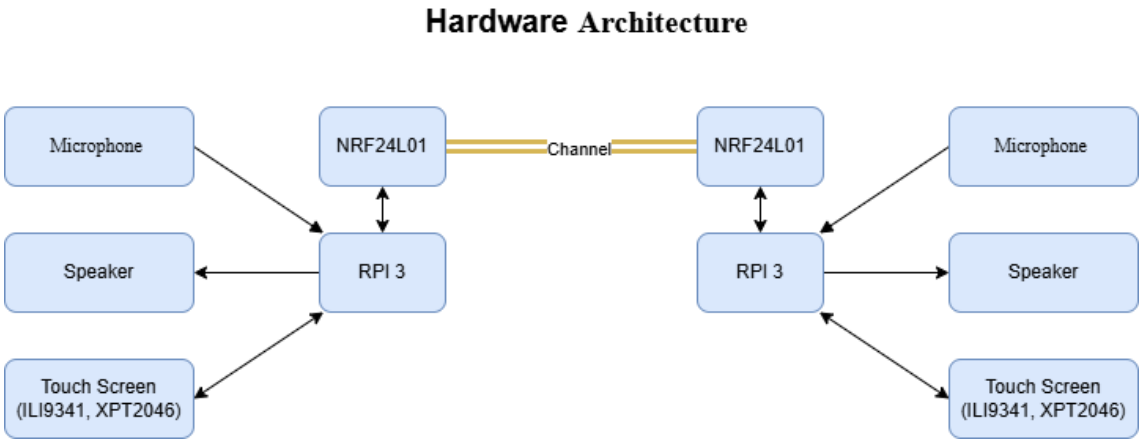


Figure 1-1: hardware architecture diagram.

The software architecture is segmented into four major layers:

1. **Control (Main) Layer** – Manages the logic flow, including command parsing, state management, and coordinating between modules.

2. **UI Layer** – Developed using LVGL in C, this layer handles all user interaction, navigation, and visual representation of message data.

3. **Communication Layer** – Encapsulates the message formatting, transmission logic, and NRF24L01 interaction using SPI.

4. **Security Layer** – Implements the RSA-based session key exchange and AES encryption decryption routines using OpenSSL, ensuring message confidentiality.

All modules communicate internally through structured APIs and shared buffers. Upon sending a message, the system triggers the security layer to encrypt data, the communication layer to transmit it, and the UI to confirm status. Similarly, incoming data is decrypted before being displayed or played back.

1.6 CONCLUSION

This chapter presented an overview of the Secure Networking Terminal project, including its background, the problems it addresses, its motivation, scope, limitations, and the methodology adopted for its implementation. The project aims to deliver a locally engineered solution for secure, offline communication by integrating embedded hardware, a touch-based user interface, voice and text messaging capabilities, and custom cryptographic routines. By targeting scenarios where conventional communication infrastructure is unavailable or untrusted, the system provides a practical tool for organizations seeking autonomous, secure communication solutions. The chapter also outlined the modular design strategy and system architecture, which form the foundation for the development and integration process detailed in the subsequent chapters.

Chapter Two

2 HARDWARE COMPONENTS

2.1 RASPBERRY PI (THE MAIN ECU)

2.1.1 Introduction

In this section, we'll talk about the main ECU in our project which is Raspberry pi. Also, we'll discuss why Raspberry pi, its Versions, Hardware Component, Interfacing with GPIO pins and Raspberry pi Operating System. The Raspberry Pi is a low-cost, credit card-sized computer that was first developed in 2012 by the Raspberry Pi Foundation in the UK. The primary goal of the Raspberry Pi was to promote the teaching of basic computer science in schools and developing countries. Initially, the Raspberry Pi was equipped with a 700 MHz ARM11 processor, 256 MB of RAM, and a single USB port. However, today's models have significantly evolved, featuring quad-core processors, up to 8GB of RAM, and multiple USB and HDMI ports.

The versatility of the Raspberry Pi extends to its ability to run a variety of operating systems, including several Linux distributions, Windows 10 IoT Core, and even Android. This flexibility allows the Raspberry Pi to be used for a wide range of projects, from simple tasks such as web browsing and playing games, to more complex endeavors like controlling robots, automating homes, and even building supercomputers.

Overall, the Raspberry Pi has become an incredibly versatile and powerful tool for anyone interested in computing and electronics.

2.1.1.1 Raspberry Pi 3 in Our Project:

For our project, we utilized the Raspberry Pi 3, which offers a significant improvement in speed and performance compared to its predecessors. The Raspberry Pi 3 provides a complete desktop experience, making it suitable for tasks such as editing documents, browsing the web with multiple tabs, managing spreadsheets, and drafting presentations. This is achieved on a smaller, more energy-efficient, and cost-effective machine.

2.1.1.2 Key Features of Raspberry Pi 3:

- **Silent and Energy-Efficient:** The Raspberry Pi 3 operates silently without a fan and consumes significantly less power than traditional computers.
- **Fast Networking:** Equipped with Gigabit Ethernet, onboard wireless networking, and Bluetooth, the Raspberry Pi 3 ensures fast and reliable connectivity.
- **SPI communication protocol:** SPI communication protocol is a must to connect the screen with raspberry pi and transceiver with raspberry pi also
- **USB 3.0:** The device includes two USB 3.0 ports, in addition to two USB 2.0 ports, allowing data transfer rates up to ten times faster than previous models.
- **Choice of RAM:** The Raspberry Pi 3 is available in different configurations, offering 1GB, 2GB, 4GB, or 8GB of RAM to meet various performance needs. These features make the Raspberry Pi 3 an excellent choice for a wide range of applications, from simple computing tasks to more complex projects involving robotics and automation.

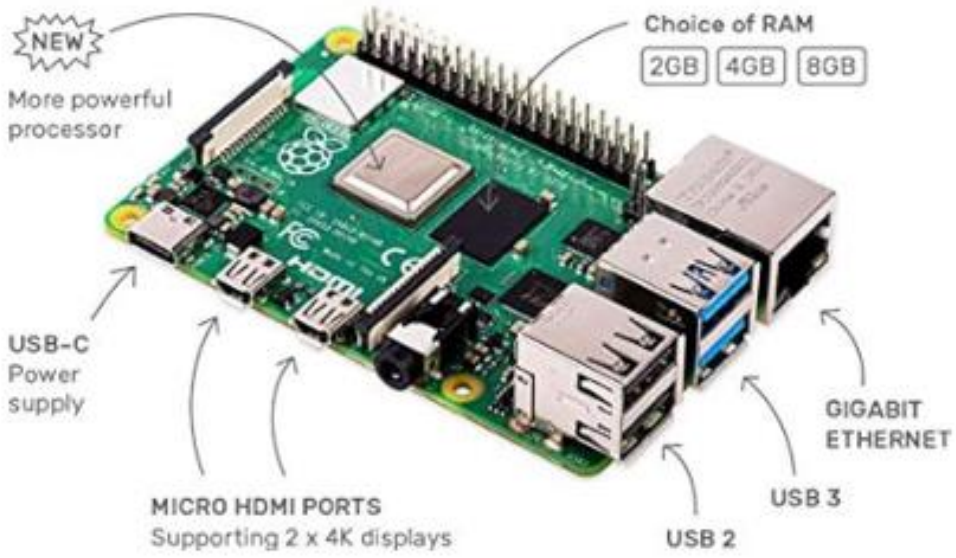


Figure 2-1: Raspberry Pi ECU

2.1.2 WHY CHOOSE RASPBERRY PI

The Raspberry Pi offers numerous advantages that make it a popular choice for a wide array of projects and applications. Here are some reasons why the Raspberry Pi is an excellent option:

- **Cost-Effective:** The Raspberry Pi is an affordable computer, making it accessible for beginners and ideal for projects with budget constraints.
- **Compact Size:** Its small and lightweight design allows for easy portability and the ability to fit into tight spaces.
- **Energy Efficient:** The Raspberry Pi consumes very little power, making it an environmentally friendly and cost-effective choice.
- **Versatility:** Capable of running various operating systems and applications, the Raspberry Pi is suitable for a broad range of projects.
- **GPIO Pins:** The General-Purpose Input/Output (GPIO) pins make it easy to connect a wide variety of devices, facilitating custom electronics projects.
- **Community Support:** A large, active community of users and developers shares knowledge and resources, making it easy to find help and solutions to problems.
- **Educational Value:** Originally designed to promote the teaching of basic computer science, the Raspberry Pi is an excellent educational tool for learning about computing and electronics.

Overall, the Raspberry Pi is a cost-effective, flexible, and versatile tool that is suitable for a wide range of projects and applications. Its popularity extends to hobbyists, students, and professionals alike, making it a go-to choice for many.

2.1.2.1 VERSIONS OF RASPBERRY PI

The Raspberry Pi has seen numerous iterations since its initial release in 2012, each version bringing improvements in processing power, memory, and connectivity options. Here's a breakdown of the major versions of the Raspberry Pi:

1. **Raspberry Pi Model B (2012)**
    - **Processor:** 700 MHz ARM11
    - **RAM:** 256 MB (later upgraded to 512 MB)
    - **Ports:** 1 USB port, HDMI, RCA video, 3.5mm audio jack
    - **Features:** The original Raspberry Pi was designed as a basic, affordable computer for educational purposes, promoting computer science in schools and developing countries.

2. Raspberry Pi Model A (2013)

- **Processor:** 700 MHz ARM11
- **RAM:** 256 MB
- **Ports:** 1 USB port, HDMI, RCA video, 3.5mm audio jack
- **Features:** A lower-cost version of the Model B, with reduced USB ports and no Ethernet port, aimed at more cost-sensitive applications.

3. Raspberry Pi Model B+ (2014)

- **Processor:** 700 MHz ARM11
- **RAM:** 512 MB
- **Ports:** 4 USB ports, HDMI, RCA video, 3.5mm audio jack, Ethernet port
- **Features:** Improved power consumption, increased GPIO pins, and better audio quality.

4. Raspberry Pi 2 Model B (2015)

- **Processor:** 900 MHz quad-core ARM Cortex-A7
- **RAM:** 1 GB
- **Ports:** 4 USB ports, HDMI, Ethernet, 3.5mm audio jack
- **Features:** Significant performance improvement with a quad-core processor and increased RAM, making it suitable for more demanding applications.

5. Raspberry Pi Zero (2015)

- **Processor:** 1 GHz single-core ARM11
- **RAM:** 512 MB
- **Ports:** Mini HDMI, 1 USB OTG port
- **Features:** Ultra-small form factor and very low cost, designed for embedded applications.

6. Raspberry Pi 3 Model B (2016)

- **Processor:** 1.2 GHz quad-core ARM Cortex-A53
- **RAM:** 1 GB
- **Ports:** 4 USB ports, HDMI, Ethernet, 3.5mm audio jack, Wi-Fi, Bluetooth
- **Features:** Built-in Wi-Fi and Bluetooth, making it more suitable for IoT applications.

7. Raspberry Pi 3 Model B+ (2018)

- **Processor:** 1.4 GHz quad-core ARM Cortex-A53
- **RAM:** 1 GB
- **Ports:** 4 USB ports, HDMI, Ethernet, 3.5mm audio jack, Wi-Fi, Bluetooth
- **Features:** Improved network speeds and thermal performance, enhanced power management.

1. **Raspberry Pi 4 Model B (2019)**

- **Processor:** 1.5 GHz quad-core ARM Cortex-A72
- **RAM:** 2GB, 4GB, or 8GB
- **Ports:** 2 USB 3.0 ports, 2 USB 2.0 ports, 2 micro-HDMI ports, Ethernet, 3.5mm audio jack, Wi-Fi, Bluetooth
- **Features:** Major upgrade with more RAM options, dual monitor support, and significantly improved CPU and GPU performance. This model is capable of providing a complete desktop experience.

2. **Raspberry Pi 400 (2020)**

- **Processor:** 1.8 GHz quad-core ARM Cortex-A72
- **RAM:** 4GB
- **Ports:** 3 USB ports, micro-HDMI ports, Ethernet, 3.5mm audio jack, Wi-Fi, Bluetooth
- **Features:** Integrated into a compact keyboard, making it a convenient all-in-one computer.

3. **Raspberry Pi Pico (2021)**

- **Processor:** Dual-core ARM Cortex-M0+
- **RAM:** 264 KB SRAM
- **Ports:** GPIO pins
- **Features:** A microcontroller rather than a full computer, designed for embedded applications, and programmable with C/C++ or Micro Python.

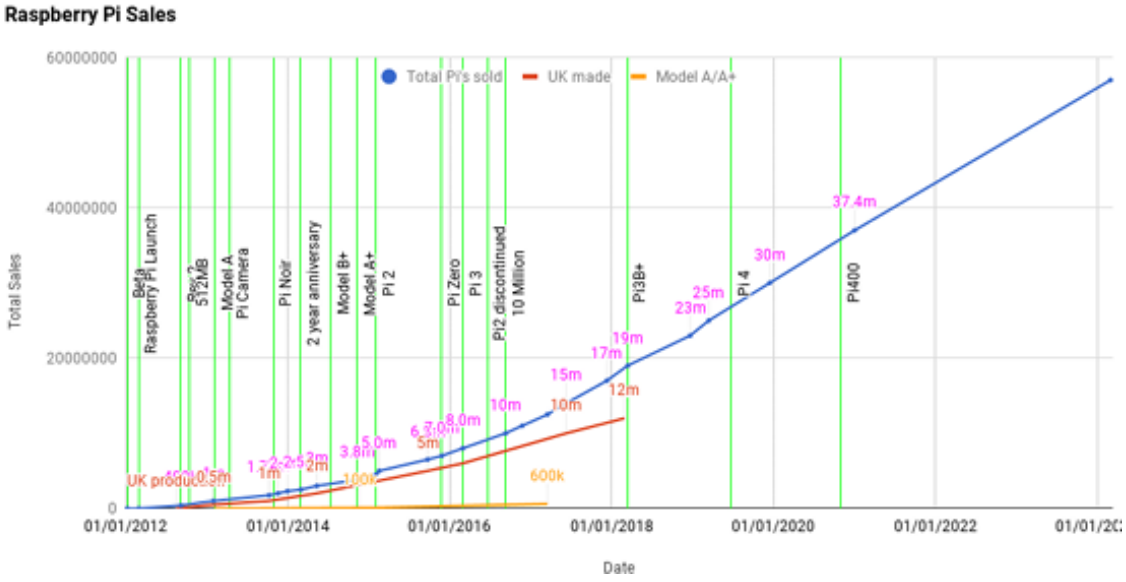


Figure 2-2: Raspberry Pi versions timeline

2.1.3 RASPBERRY PI HARDWARE

2.1.3.1 Overview:

In this section, we will discuss the hardware aspects of the Raspberry Pi 3, including its specifications, pin diagram, and descriptions. We will also cover the layout and interfacing of GPIO pins, as well as the various hardware components used in the Raspberry Pi.

2.1.3.2 Hardware Components of Raspberry Pi

The hardware components of the Raspberry Pi include:

- **CPU/GPU:** The ARM Cortex-A72 CPU and Video Core VI GPU provide the processing power and graphical capabilities.
- **RAM:** Depending on the model, the Raspberry Pi 3 can have 2GB, 4GB, or 8GB of RAM.
- **Networking:** Integrated Gigabit Ethernet, Wi-Fi, and Bluetooth enable robust connectivity options. 21
- **USB Ports:** USB 3.0 and USB 2.0 ports for connecting peripherals and storage devices.

- **HDMI Ports:** Dual micro-HDMI ports allow for dual monitor setups with up to 4K resolution.
- **Power Supply:** The USB-C power supply ensures sufficient power for all components and peripherals.

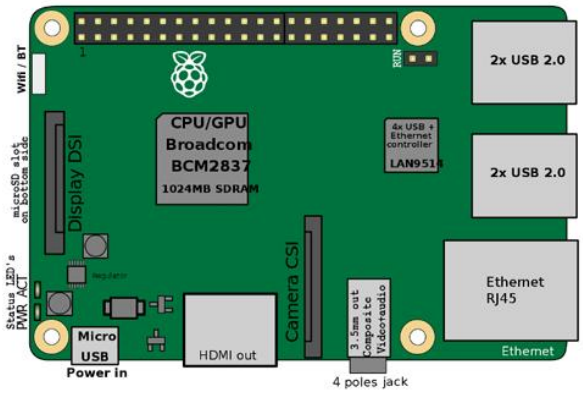


Figure 2-3: Raspberry Pi hardware components

Figure 3-1-3 explain the position of each component, these components work together to provide a versatile and powerful computing platform suitable for a wide range of applications, from basic computing tasks to complex electronics projects and automation systems.

2.1.3.3 Raspberry Pi 3 Specifications

The Raspberry Pi 3 represents a significant upgrade over previous models, offering enhanced performance and connectivity options. Here are some key specifications:

- **Processor:** 1.5 GHz quad-core ARM Cortex-A72
- **RAM:** Options of 2GB, 4GB, and 8GB
- **Networking:** Gigabit Ethernet, Wi-Fi 802.11ac, Bluetooth 5.0 22
- **USB Ports:** 2 USB 3.0 ports and 2 USB 2.0 ports
- **Video Output:** 2 micro-HDMI ports supporting up to 4K resolution
- **Storage:** MicroSD card slot for storage and operating system
- **Power:** USB-C power supply

2.1.3.4 Layout and Interfacing of GPIO Pins

The pin diagram of the Raspberry Pi 3 includes 40 GPIO pins, which can be used for various purposes such as digital input/output, PWM, I2C, SPI, and UART.
The GPIO (General Purpose Input/Output) pins on the Raspberry Pi 3 are a key feature that allows users to interface with a wide range of sensors, actuators, and other electronic components. The GPIO header consists of 40 pins, arranged in a 2x20 grid, each with specific functions:

- **Power Pins**: Provide 3.3V and 5V power to connected devices.
- **Ground Pins:** Common ground for all connected devices.
- **GPIO Pins:** Configurable pins that can be used for digital input/output.
- **Special Function Pins:** Dedicated pins for communication protocols like I2C, SPI, and UART.

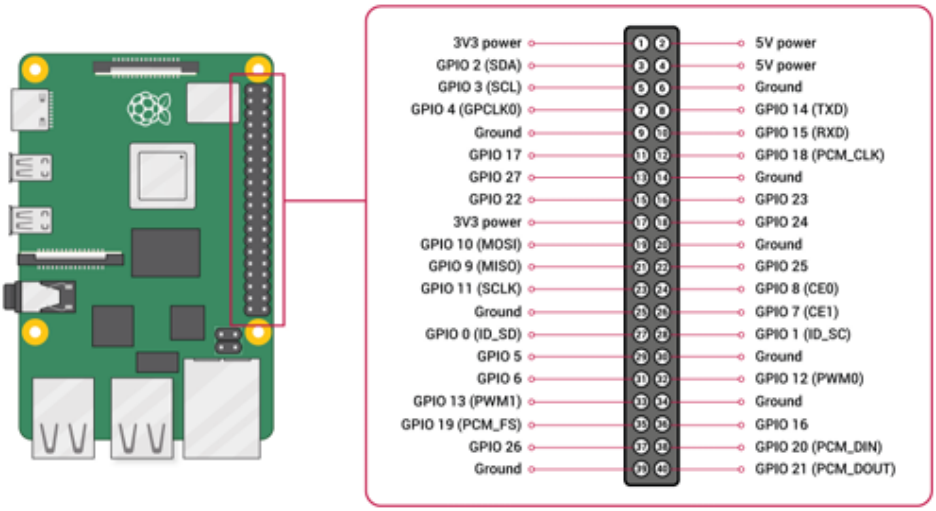


Figure 2-4: Raspberry Pi Pin Diagram

2.1.3.5 RASPBERRY PI OS

Overview

Raspberry Pi OS, formerly known as Raspbian, is the official operating system for the Raspberry Pi. It is a free and open-source OS based on the Debian Linux distribution, specifically optimized for the Raspberry Pi hardware. Raspberry Pi OS comes pre-installed with a variety of software to help users get started with their projects.

Key Features and Components

- **Desktop Environment**
  - **Description**: Raspberry Pi OS features a desktop environment based on the Lightweight X11 Desktop Environment (LXDE), which is optimized for the Raspberry Pi's hardware.
  - **Benefit**: Provides a familiar and user-friendly interface for performing various tasks such as browsing the web, editing documents, and managing files.

- Pre-installed Software
  - **Description**: The OS includes a variety of pre-installed software, such
    - **Chromium Web Browser**: Useful for testing web interfaces, accessing documentation, and downloading libraries or tools directly on the Raspberry Pi.
    - **Terminal (LXTerminal)**: Allows you to compile, run, and debug your C code. Useful for managing network connections and running system commands.
    - **Thonny Python IDE**: Even though your main project is in C, you can use Thonny for quick scripts (e.g., testing communication protocols, simulating server/client behavior).
    - **VNC Viewer and Remote Desktop Tools**: Useful for remotely accessing and demonstrating your GUI interface on the Raspberry Pi screen.
    - **Geany Text Editor**: Lightweight editor for writing and editing your C driver files and configuration scripts.
    - **File Manager**: Allows you to organize your project files, move data, and manage dependencies easily without command line.

- **Bluetooth Manager**: Can be used if your secure terminal includes Bluetooth communication.

- **Wi-Fi Configuration Tool**: Helps set up and test network connectivity, which is essential for secure communication features.

- **Raspi-Config Tool**: Used to enable SPI interface (required for ILI9341 and XPT2046), SSH access, and other hardware configurations.

- **Pre-Installed Compilers and Build Tools (GCC, Make)**: Essential for compiling and building your C applications directly on the Raspberry Pi.

- Package Manager
  - Description: The APT package manager is included, which allows users to install new software packages and keep the system up to date.
  - Benefit: Ensures that users can easily extend the functionality of their Raspberry Pi and maintain security and performance updates.

- GPIO Support
  - Description: Raspberry Pi OS supports the GPIO (General Purpose Input/Output) pins on the Raspberry Pi, allowing users to interface with external hardware and sensors.
  - Benefit: Facilitates hardware projects, such as robotics and home automation, by providing direct access to the GPIO pins for programming and control.

Conclusion Overall, Raspberry Pi OS is a powerful and flexible operating system tailored for the Raspberry Pi hardware. It is an excellent choice for a wide range of projects, from building a media centre to creating a home automation system. The OS's combination of a user-friendly desktop environment, pre-installed software, and robust configuration tools makes it ideal for both beginners and advanced users.

2.2 TOUCH SCREEN

Figure 2-5: (a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color)

2.2.1 Introduction to the ILI9341 Display Driver

The ILI9341 is a sophisticated 262,144-color single-chip System-on-Chip (SoC) driver meticulously engineered for a-Si (amorphous silicon) TFT liquid crystal displays. It is designed to manage displays with a resolution of 240RGBx320 dots, integrating crucial display functionalities directly onto a single die. This integration includes a 720-channel source driver, a 320-channel gate driver, and a substantial 172,800 bytes of Graphics RAM (GRAM) dedicated to storing display data for the 240RGBx320 dot resolution. This on-chip memory is a key differentiator, as it eliminates the need for external display RAM, simplifying system design and reducing overall component count. The ILI9341's comprehensive feature set, encompassing flexible interface options, robust power management, and advanced display control, positions it as an ideal solution for a diverse range of portable electronic products. These include, but are not limited to, digital cellular phones, smartphones, MP3 players, and Portable Multimedia Players (PMPs), where optimizing battery life and delivering vibrant visual experiences are paramount considerations.

2.2.2 Key Features and Architectural Overview

The ILI9341's design is characterized by a rich array of features and a well-defined internal architecture that collectively enable its high performance and versatility:

2.2.2.1 Display Capabilities

- **Resolution and Color Depth**: The driver natively supports a display resolution of 240 horizontal RGB dots by 320 vertical dots. It can render up to 262,144 full colors, offering a rich visual experience. A "reduce color mode" (8-color) is also available, specifically optimized for "Idle Mode ON" to conserve power when full color fidelity is not required.

- **On-chip Display RAM (GRAM)**: A significant 172,800 bytes of GRAM are embedded within the chip, providing ample space for storing 240RGBx320 dots of graphic display data. This integrated memory is crucial for enabling the "window address function," which allows for selective updating of specified moving picture areas within the GRAM. This functionality facilitates the simultaneous display of dynamic content alongside static images without needing to refresh the entire screen, thereby improving efficiency and responsiveness.

2.2.2.2 System Interface Options

The ILI9341 offers extensive compatibility with various host controllers through multiple system interfaces:

- **MCU Parallel Interfaces**: It provides support for 8-bit, 9-bit, 16-bit, and 18-bit data bus interfaces, compatible with both 8080-I and 8080-II series Microcontroller Units (MCUs). This broad compatibility allows for seamless integration into diverse embedded systems.

- **RGB Interfaces**: For applications requiring direct display rendering from graphics controllers, the ILI9341 supports 6-bit, 16-bit, and 18-bit data bus RGB interfaces. This is particularly beneficial for displaying high-frame-rate content.

- **Serial Peripheral Interface (SPI)**: For reduced pin count and simplified wiring, 3-line and 4-line serial interfaces are available.

2.2.2.3 Power Management and On-Chip Functionality

The driver is designed with a strong emphasis on power efficiency, crucial for portable devices:

- **Power Saving Modes**: It includes "Sleep Mode" and "Deep Standby Mode" to minimize power consumption when the display is not actively being used.

- **Voltage Requirements**: The I/O interface operates from 1.65V to 3.3V, while the analog supply voltage (VCI) ranges from 2.5V to 3.3V. An incorporated voltage follower circuit efficiently generates the necessary voltage levels for driving the LCD panel.

- **Integrated Circuits**: Key functional blocks integrated on the chip include:
  - **VCOM Generator and Adjustment**: Generates and fine-tunes the common electrode voltage (VCOM).
  - **Timing Generator**: Produces all essential timing signals for display operation and GRAM access.
  - **Oscillator**: An on-chip RC oscillator provides a stable output frequency for internal operations.
  - **DC/DC Converter**: Generates the various voltage levels (GVDD, VGH, VGL) required for TFT LCD panel driving.
  - **Line/Frame Inversion**: Supports inversion techniques to prevent image sticking and improve display uniformity.
  - **Gamma Correction**: Features a preset Gamma curve with separate RGB Gamma correction capabilities for accurate color reproduction.
  - **Content Adaptive Brightness Control (CABC)**: Dynamically adjusts display brightness based on image content, further enhancing power efficiency.

2.2.2.4 Panel Driver Circuitry

The core of the display driving capability resides in its integrated panel driver circuits:

- **Source Driver**: Comprises 720 output channels (S1 to S720).
- **Gate Driver**: Consists of 320 output channels (G1 to G320).
- **VCOM Signal**: Provides the common electrode signal for the TFT display.

2.2.3 Detailed MCU and RGB Interface Functionality

The ILI9341's versatile interface options are crucial for its broad applicability.

2.2.3.1 MCU Interfaces

The driver offers both parallel and serial interfaces to communicate with a host MCU. The selection between these modes and their specific configurations is determined by the external IM [3:0] pins.

Table 2-1 : interface table 8080-I and 8080-II Series Parallel Interfaces

IM3	IM2	IM1	IM0	MCU-Interface Mode	Pins in use	
					Register/Content	GRAM
0	0	0	0	8080 MCU 8-bit bus interface Ⅰ	D[7:0]	D[7:0],WRX,RDX,CSX,D/CX
0	0	0	1	8080 MCU 16-bit bus interface Ⅰ	D[7:0]	D[15:0] ,WRX,RDX,CSX,D/CX
0	0	1	0	8080 MCU 9-bit bus interface Ⅰ	D[7:0]	D[8:0] ,WRX,RDX,CSX,D/CX
0	0	1	1	8080 MCU 18-bit bus interface Ⅰ	D[7:0]	D[17:0] ,WRX,RDX,CSX,D/CX
0	1	0	1	3-wire 9-bit data serial interface Ⅰ	SCL,SDA,CSX	
0	1	1	0	4-wire 8-bit data serial interface Ⅰ	SCL,SDA,D/CX,CSX	
1	0	0	0	8080 MCU 16-bit bus interface Ⅱ	D[8:1]	D[17:10],D[8:1],WRX,RDX,CSX,D/CX
1	0	0	1	8080 MCU 8-bit bus interface Ⅱ	D[17:10]	D[17:10],WRX,RDX,CSX,D/CX
1	0	1	0	8080 MCU 18-bit bus interface Ⅱ	D[8:1]	D[17:0],WRX,RDX,CSX,D/CX
1	0	1	1	8080 MCU 9-bit bus interface Ⅱ	D[17:10]	D[17:9],WRX,RDX,CSX,D/CX
1	1	0	1	3-wire 9-bit data serial interface Ⅱ	SCL,SDI,SDO, CSX	
1	1	1	0	4-wire 8-bit data serial interface Ⅱ	SCL,SDI,D/CX,SDO, CSX	

Table 2-3: 8080 I series parallel interface table

IM3	IM2	IM1	IM0	MCU-Interface Mode	CSX	WRX	RDX	D/CX	Function
0	0	0	0	8080 MCU 8-bit bus interface Ⅰ	"L"	↑	"H"	"L"	Write command code.
					"L"	"H"	↑	"H"	Read internal status.
					"L"	↑	"H"	"H"	Write parameter or display data.
					"L"	"H"	↑	"H"	Reads parameter or display data.
0	0	0	1	8080 MCU 16-bit bus interface Ⅰ	"L"	↑	"H"	"L"	Write command code.
					"L"	"H"	↑	"H"	Read internal status.
					"L"	↑	"H"	"H"	Write parameter or display data.
					"L"	"H"	↑	"H"	Reads parameter or display data.
0	0	1	0	8080 MCU 9-bit bus interface Ⅰ	"L"	↑	"H"	"L"	Write command code.
					"L"	"H"	↑	"H"	Read internal status.
					"L"	↑	"H"	"H"	Write parameter or display data.
					"L"	"H"	↑	"H"	Reads parameter or display data.
0	0	1	1	8080 MCU 18-bit bus interface Ⅰ	"L"	↑	"H"	"L"	Write command code.
					"L"	"H"	↑	"H"	Read internal status.
					"L"	↑	"H"	"H"	Write parameter or display data.
					"L"	"H"	↑	"H"	Reads parameter or display data.

2.2.3.2 Signal Lines: Common control signals include:

- **CSX (Chip Select)**: Active low, used to enable or disable the ILI9341 chip.
- **RESX (Reset)**: Active low, an external reset signal.

- **WRX (Write Strobe)**: Data is latched on the rising edge of this signal.
- **RDX (Read Strobe)**: Used for reading data from the driver, with data being read by the MCU on its rising edge.
- **D/CX (Data/Command Select)**: A crucial signal that determines whether the data on the D[17:0] bus is a command (D/CX = '0') or display RAM data/command parameters (D/CX = '1').
- **Data Bus**: The D[17:0] pins serve as the bi-directional parallel data bus.

Write Cycle Sequence: During a write cycle, the host asserts the data on the D[17:0] lines, and the ILI9341 captures this data on the rising edge of WRX. The D/CX signal indicates whether the data is a command or parameters/display data.

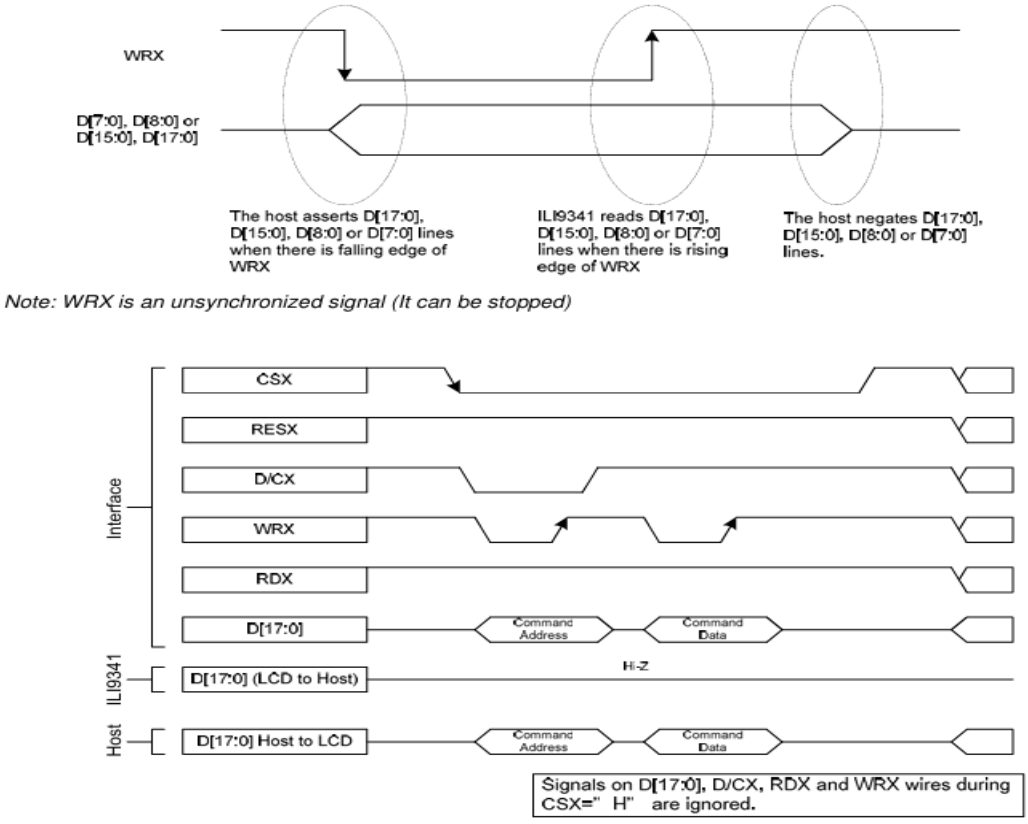


Figure 2-6: write cycle for the 8080 I

Read Cycle Sequence: In a read cycle, the RDX signal is driven low and then high. The ILI9341 asserts the requested data on D[17:0] on the falling edge of RDX, and the host reads it on the rising edge. Read data is only valid when D/CX is high; otherwise, the outputs are High-Z.

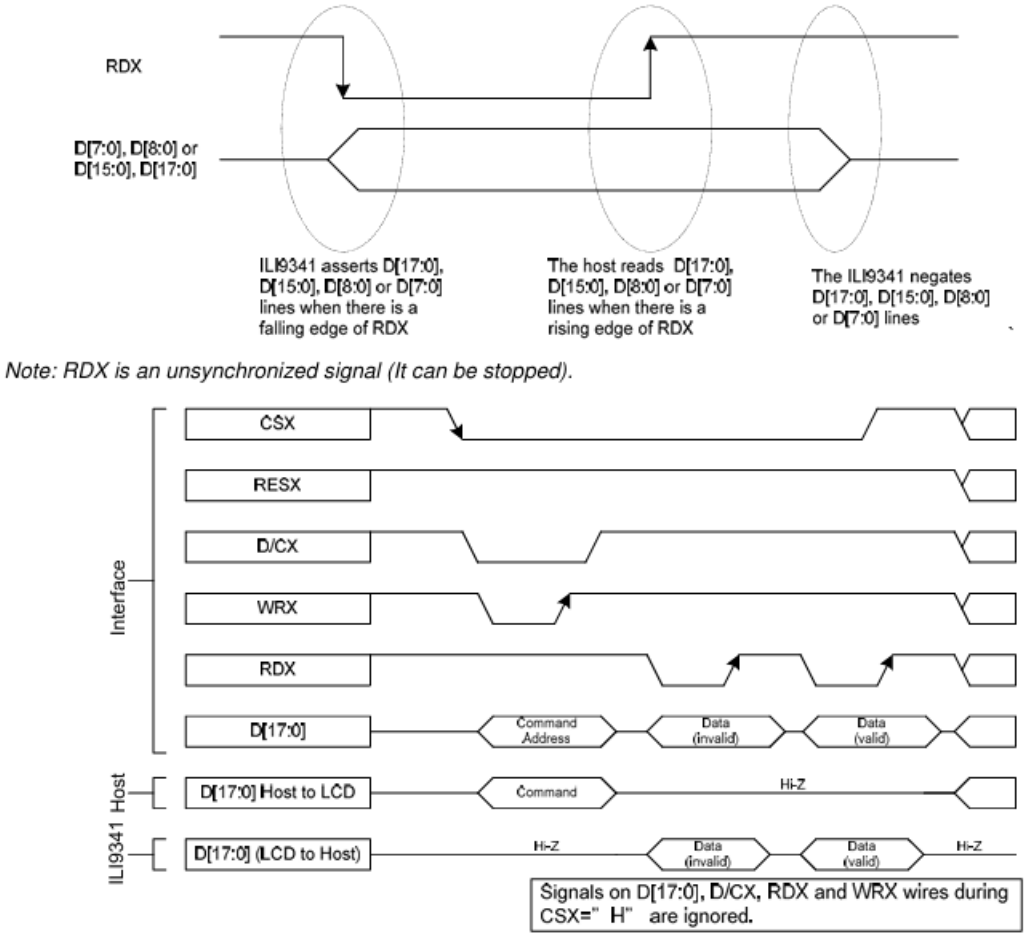


Figure 2-7: read cycle for the 8080 I

2.2.3.3 Serial Interface (SPI)

Table 2-5: series interface table

IM3	IM2	IM1	IM0	MCU-Interface Mode	CSX	D/CX	SCL	Function
0	1	0	1	3-line serial interface	"L"	-	↑	Read/Write command, parameter or display data.
0	1	1	0	4-line serial interface	"L"	'H/L"	↑	Read/Write command, parameter or display data.
1	1	0	1	3-line serial interface	"L"	-	↑	Read/Write command, parameter or display data.
1	1	1	0	4-line serial interface	"L"	'H/L"	↑	Read/Write command, parameter or display data.

The ILI9341 supports both 3-line/9-bit and 4-line/8-bit bi-directional serial interfaces.

- **3-line SPI**: Utilizes CSX, SCL (Serial Clock), and a single SDA (Serial Data Input/Output) line. Data packets include a Data/Command select bit within the

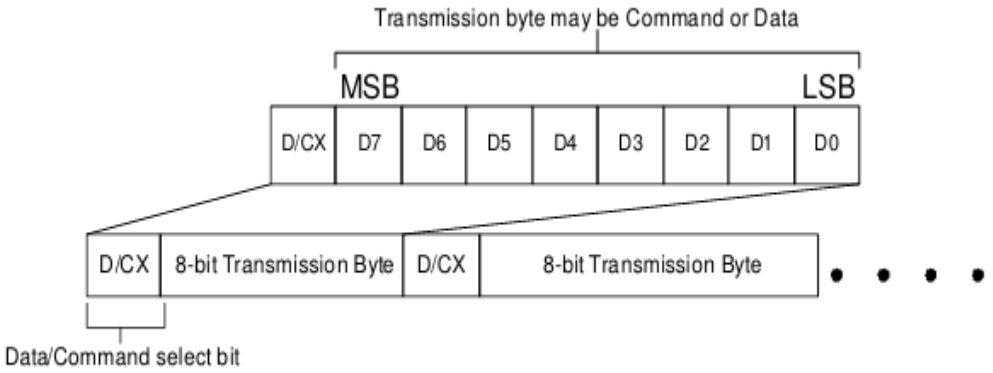


Figure 2-8: 3-line SPI sequence transmission byte.

- **4-line SPI**: Consists of CSX, SCL, a separate SDI (Serial Data Input) and SDO (Serial Data Output), and a dedicated D/CX pin.

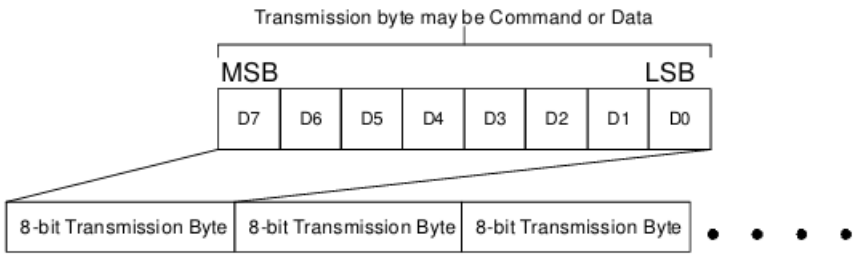


Figure 2-9: 4-line SPI sequence

- **Data Transmission**: The Most Significant Bit (MSB) is transmitted first. Data is latched on the rising edge of SCL, and output data is shifted on the falling edge of SCL. The serial clock can be stopped when no communication is necessary.

2.2.4 RGB Interface

The RGB interface is ideal for applications requiring high-speed display updates, such as video. When using the RGB interface, the serial interface must be selected for register configuration. The RCM[1:0] bits determine the specific RGB interface mode.

2.2.4.1 RGB Interface Modes

- **DE Mode (Data Enable)**: Selected when RCM[1:0] is set to "10". In this mode, display operation is synchronized with external VSYNC, HSYNC, and DOTCLK signals. Display data is transferred to the internal GRAM in synchronization with these signals, governed by the active-high DE (Data Enable) signal.

- **SYNC Mode (Synchronization)**: Selected when RCM[1:0] is set to "11". In this mode, the DE signal is ignored, and blanking porch periods are defined by the B5h command. Valid display data is input in pixel units via the D[17:0] pins according to the HFP/HBP (Horizontal Front/Back Porch) and VFP/VBP (Vertical Front/Back Porch) settings of the HSYNC and VSYNC signals, respectively.

2.2.4.2 Pixel Formats

The RGB interface supports several pixel formats, chosen by the DPI[2:0] bits of the "Pixel Format Set (3Ah)" command and the RIM bit of the F6h command:

- **18-bit RGB interface (262K colors)**: Uses D[17:0].
- **16-bit RGB interface (65K colors)**: Uses D[17:13] & D[11:1]. The LSB data of red/blue colors can depend on the EPF[1:0] setting.
- **6-bit RGB interface (262K/65K colors)**: Uses D[5:0]. In this mode, each dot of one pixel (R, G, and B) is transferred sequentially in synchronization with DOTCLK.

2.2.4.3 Timing Signals

- **DOTCLK (Pixel Clock)**: Runs continuously and is used to sample VSYNC, HSYNC, DE, and D[17:0] states on its rising edge.

- **VSYNC (Vertical Synchronization)**: Active low, indicates the start of a new display frame.
- **HSYNC (Horizontal Synchronization)**: Active low, indicates the start of a new display line.
- **DE (Data Enable)**: Active high in DE mode, signifies valid RGB information.

2.2.4.4 VSYNC Interface

The VSYNC interface is employed in conjunction with the 8080-I/8080-II system interface to display moving pictures. This mode synchronizes the display operation with the internal clock and the external VSYNC signal, where the VSYNC signal dictates the frame rate. Display data is stored in the GRAM, minimizing data transfer for dynamic content. Critical considerations include meeting minimum GRAM update speeds and ensuring the VSYNC period is longer than the entire display scan period. Partial display, vertical scroll, and interlaced scan functions are not available in VSYNC interface mode.

2.2.5 Display Data RAM (DDRAM) Configuration

The ILI9341 integrates a static RAM dedicated to display data, totaling 1,382,400 bits (240 columns x 18 bits/pixel x 320 rows). This substantial on-chip memory ensures that the display can operate autonomously without constant external data feeding. A key advantage of this architecture is the absence of abnormal visible effects on the display even during simultaneous panel display reads and interface read/write operations to the same memory location. This means the host processor can update display content in the background while the display is actively refreshing, leading to smoother and more efficient graphics updates.

The organization of the DDRAM directly impacts how pixels are mapped to the display. In "Normal Display ON" or "Partial Mode ON" with "Vertical Scroll Mode OFF," the display area corresponds directly to column pointers 0000h to 00EFh and page pointers 0000h to 013Fh in the frame memory. This means storing data at (column, page) = (0, 0) will display it at the top-left corner of the screen.
For "Vertical Scroll Mode," the commands "Vertical Scrolling Definition" (33h) and "Vertical Scrolling Start Address" (37h) define the scrolling area and behavior. This allows for dynamic vertical movement of content on the display, crucial for applications with long lists or continuous data streams. The memory-to-display address mapping adapts to these scrolling definitions, ensuring correct content presentation during vertical shifts.

The MCU to memory write/read direction can be controlled by the "Memory Access Control" (MADCTL) command (36h). This command's bits (B5, B6, and B7) dictate how the incoming data stream from the MCU is written into the physical memory, allowing for various display orientations (e.g., normal, X-mirror, Y-mirror, XY-exchange) without altering the data stream itself. Regardless of these settings, data is always written to the Frame Memory in a consistent pixel-unit order.

2.2.6 Color Depth Conversion

The ILI9341 efficiently handles different color depths to accommodate varying input data formats while maintaining its 18-bit display capability.

- **16-bit to 18-bit Conversion**: When the ILI9341 operates with a 16-bit parallel interface (e.g., RGB 5-6-5 format for 65,536 colors), an internal look-up table (LUT) is employed to expand the 16-bit input data to the native 18-bit format (RGB 6-6-6) used by the display. This ensures that even with a reduced input color depth, the full 262,144 color palette of the display can be utilized by intelligently mapping the available input values to the wider output range. The "Color Set" command (2Dh) is specifically used to define this LUT, requiring 128 bytes of data to be written.
- **Direct 18-bit Input**: For applications providing 18-bit data directly, the driver accepts RGB 6-6-6 input, bypassing the LUT conversion.

2.2.7 3.2.6 Command Set Reference

The ILI9341 features a comprehensive command set, categorized into "Regulative Command Set" (Level 1) and "Extended Command Set" (Level 2). These commands provide granular control over every aspect of the display driver's operation, from basic power states to advanced gamma correction.

2.2.7.1 Level 1 Commands

These commands are fundamental for controlling the display and accessing its memory.

- **System Operations**:
  - **NOP (00h)**: A no-operation command that can terminate frame memory write or read operations.
  - **Software Reset (01h)**: Resets commands and parameters to their default values, but does not affect frame memory contents.
  - **Sleep OUT (11h) / Enter Sleep Mode (10h)**: Controls the power-saving modes, enabling or disabling DC/DC converters, internal oscillators, and panel

scanning. A significant delay (120ms) is required after Sleep Out to allow circuits to stabilize.

- **Display Modes**:
  - **Display ON (29h) / Display OFF (28h)**: Enables or disables the display output from frame memory.
  - **Normal Display Mode ON (13h) / Partial Mode ON (12h)**: Toggles between full display and a defined partial display area. The partial area is defined by the Partial Area command (30h).
  - **Display Inversion ON (21h) / Display Inversion OFF (20h)**: Inverts all pixel data from the frame memory to the display, without changing memory contents.
  - **Idle Mode ON (39h) / Idle Mode OFF (38h)**: Reduces color expression to 8 primary/secondary colors in "Idle Mode ON" for power saving.

- **Memory Access and Addressing**:
  - **Column Address Set (2Ah) / Page Address Set (2Bh)**: Defines the accessible area of the frame memory for MCU read/write operations.
  - **Memory Write (2Ch)**: Transfers data from the MCU to the frame memory, starting from the defined column/page addresses.
  - **Memory Read (2Eh)**: Transfers image data from frame memory to the host processor.
  - **Write_Memory_Continue (3Ch) / Read_Memory_Continue (3Eh)**: Allows for sequential writing or reading of image data from the frame memory without re-specifying the address.
  - **Memory Access Control (36h)**: Defines the read/write scanning direction of the frame memory (e.g., row/column order, refresh direction, RGB/BGR order).

- **Display Information and Status**:
  - **Read display identification information (04h)**: Returns manufacturer, version, and module/driver ID.
  - **Read Display Status (09h)**: Provides current status of booster, address order, display modes (Idle, Partial, Sleep), and other display parameters.
  - **Read Display Power Mode (0Ah)**: Indicates the status of booster, idle mode, partial mode, sleep mode, and display on/off.
  - **Read Display MADCTL (0Bh)**: Returns the current memory access control settings (row/column order, exchange, refresh direction, RGB/BGR).
  - **Read Display Pixel Format (0Ch)**: Shows the current pixel format settings for both RGB and MCU interfaces.
  - **Read Display Image Format (0Dh)**: Indicates the currently selected gamma curve.

- **Read Display Signal Mode (0Eh)**: Provides status of tearing effect line, horizontal/vertical sync, pixel clock, and data enable signals for the RGB interface.
- **Read Display Self-Diagnostic Result (0Fh)**: Indicates if register loading and general display functionality are working correctly.

- **Tearing Effect Control**:
  - **Tearing Effect Line OFF (34h) / Tearing Effect Line ON (35h)**: Controls the Tearing Effect output signal (TE pin), used for synchronizing MCU to frame writing.
  - **Set_Tear_Scanline (44h)**: Turns on the TE signal when the display reaches a specified line.
  - **Get_Scanline (45h)**: Returns the current scanline being updated on the display.

- **Brightness Control**:
  - **Write Display Brightness (51h)**: Adjusts the overall brightness level of the display.
  - **Read Display Brightness (52h)**: Returns the current brightness value.
  - **Write CTRL Display (53h)**: Controls brightness block, display dimming, and backlight on/off.
  - **Read CTRL Display (54h)**: Returns the current brightness control settings.
  - **Write Content Adaptive Brightness Control (55h)**: Sets parameters for CABC modes (Off, User Interface, Still Picture, Moving Image).
  - **Read Content Adaptive Brightness Control (56h)**: Reads the current CABC mode settings.
  - **Write CABC Minimum Brightness (5Eh)**: Sets the minimum brightness level for CABC function to prevent excessive dimming.
  - **Read CABC Minimum Brightness (5Fh)**: Reads the configured minimum brightness for CABC.

- **ID Reading**:
  - **Read ID1 (DAh) / Read ID2 (DBh) / Read ID3 (DCh)**: Used to identify the LCD module's manufacturer, version, and driver ID.

2.2.7.2 Level 2 Commands

These commands offer more advanced and specific controls, often related to factory calibration or fine-tuning. These commands typically require the EXTC pin to be high to be enabled.

- **RGB Interface Signal Control (B0h)**: Sets the operation status of the display interface, including polarity settings for DOTCLK, HSYNC, VSYNC, and DE signals, as well as RGB interface selection (RCM[1:0]) and bypass mode.

- **Frame Rate Control (B1h, B2h, B3h)**:
  - **FRMCTR1 (B1h)**: Controls frame rate in normal mode (full colors).
  - **FRMCTR2 (B2h)**: Controls frame rate in idle mode (8 colors).
  - **FRMCTR3 (B3h)**: Controls frame rate in partial mode (full colors). These commands define division ratios (DIVA/B/C) and clock cycles per line (RTNA/B/C) to calculate the display's frame frequency.

- **Display Inversion Control (B4h)**: Sets inversion mode (line or frame inversion) for different display modes (normal, idle, partial).

- **Blanking Porch Control (B5h)**: Defines the line numbers for vertical front/back porch (VFP/VBP) and horizontal front/back porch (HFP/HBP) periods in dot clocks.

- **Display Function Control (B6h)**: Controls various display functions including scan mode in non-display areas (PTG), source/VCOM output in non-display areas (PT), source driver shift direction (SS), liquid crystal type (REV), gate driver scan direction (GS), gate driver pin arrangement (SM), and number of lines to drive LCD (NL). It also includes PCDIV for external oscillator frequency division.

- **Entry Mode Set (B7h)**: Controls deep standby mode (DSTB), low voltage detection (GAS), and gate driver output levels (GON/DTE).

- **Backlight Control (B8h, B9h, BAh, BBh, BCh, BEh, BFh)**: A suite of commands for fine-tuning backlight behavior, including:
  - **B8h (Backlight Control 1)**: Sets percentage of grayscale data accumulate histogram value in user interface (UI) mode (TH_UI).
  - **B9h (Backlight Control 2)**: Sets histogram values for still picture (TH_ST) and moving image (TH_MV) modes.
  - **BAh (Backlight Control 3)**: Sets minimum grayscale threshold in UI mode (DTH_UI).

- **BBh (Backlight Control 4)**: Sets minimum grayscale threshold for still picture (DTH_ST) and moving image (DTH_MV) modes.
- **BCh (Backlight Control 5)**: Sets brightness transition time (DIM1) and brightness change threshold (DIM2).
- **BEh (Backlight Control 7)**: Controls PWM output frequency (PWM_DIV) for backlight.
- **BFh (Backlight Control 8)**: Defines polarity of LEDPWM and LEDON signals.

- **Power Control (C0h, C1h)**:
  - **PWCTRL 1 (C0h)**: Sets GVDD level (VRH), a reference for VCOM and grayscale voltages.
  - **PWCTRL 2 (C1h)**: Sets the step-up factor (BT) for internal power supply circuits.

- **VCOM Control (C5h, C7h)**:
  - **VMCTRL1 (C5h)**: Sets VCOMH and VCOML voltages.
  - **VMCTRL2 (C7h)**: Enables VCOM offset adjustment (nVM) and sets the VCOM offset voltage (VMF).

- **NV Memory Operations**:
  - **NV Memory Write (D0h)**: Programs data to NV memory for ID1, ID2, ID3, and VMF settings.
  - **NV Memory Protection Key (D1h)**: Requires a specific key (0x55AA66h) to enable NV memory programming.
  - **NV Memory Status Read (D2h)**: Reads the program record count for NV memory items and programming busy status.

- **Read ID4 (D3h)**: Reads the IC device code, including IC version and model name.

- **Gamma Correction (E0h, E1h, E2h, E3h)**:
  - **Positive Gamma Control (E0h)**: Sets grayscale voltage points for positive gamma correction.
  - **Negative Gamma Correction (E1h)**: Sets grayscale voltage points for negative gamma correction.

- **Digital Gamma Control 1 (E2h)**: Provides macro-adjustment registers for red and blue gamma curves.
- **Digital Gamma Control 2 (E3h)**: Provides micro-adjustment registers for red and blue gamma curves.

- **Interface Control (F6h)**: Selects 16-bit data format, data transfer method (MDT), endianness (ENDIAN), memory write control (WEMODE), display operation mode (DM), RAM access interface (RM), and RGB interface mode (RIM).

2.2.8 Power On/Off and Reset Sequences

Proper power-on/off and reset sequences are critical for the stable and reliable operation of the ILI9341. The datasheet outlines specific timing requirements to ensure correct initialization and power cycling.

1 **Power Supply Order**: VDDI and VCI can be applied and powered down in any order.

2 **RESX Control**: If RESX is held high or unstable during power-on, a hardware reset is mandatory after VCI and VDDI are stable. If RESX is held low during power-on, it must remain low for at least 10μs after power supplies are stable

3 **Power Off**: Different power-down timings apply depending on whether the LCD is in Sleep Out or Sleep In mode.

4 **Uncontrolled Power Off**: The design ensures no damage to the display module or host during an uncontrolled power-off event. The display will blank within 1 second and remain blank until a proper Power on Sequence.

5 **Reset Timings**: A reset pulse (RESX low) duration of at least 10μs is required for a valid reset. A "reset cancel" period (tRT) is also specified, allowing time for internal loading of ID bytes, VCOM settings, and other factory defaults.

6 **Register Initialization**: Upon power-on, hardware reset, or software reset, specific registers are set to their default values, while frame memory contents remain unaffected by software or hardware resets.

2.2.9 Power Level Definitions

The ILI9341 defines seven distinct power level modes, ranging from maximum to minimum power consumption, offering granular control for optimized battery life.

1. **Normal Mode On (Full Display), Idle Mode Off, Sleep Out**:
   Maximum power, 262,144 colors.

2. **Partial Mode On, Idle Mode Off, Sleep Out**:
   Partial display area, 262,144 colors.

3. **Normal Mode On (Full Display), Idle Mode On, Sleep Out**:
   Full display area, 8 colors.

4. **Partial Mode On, Idle Mode On, Sleep Out**:
   Partial display area, 8 colors.

5. **Sleep In Mode**:
   DC/DC converter, internal oscillator, and panel driver stopped. MCU interface and memory remain active, preserving memory contents.

6. **Deep Standby Mode**:
   Internal logic and SRAM power are off; display data and instructions are lost. Exit requires a specific CSX sequence or RESX pulse.

7. **Power Off Mode**:
   Both VCI and VDDI removed

2.3 ZAFFIRO USB DESKTOP MICROPHONE M-DU02

2.3.1 Technical Overview of the ZAFFIRO USB Microphone

The ZAFFIRO M-DU02 is a **condenser USB microphone** specifically designed for desktop use. Its main features include:

- **Plug-and-play USB interface**: No driver installation is needed, ensuring compatibility with most Linux-based systems including Raspberry Pi OS.
- **Omnidirectional pickup pattern**: Captures sound from all directions, which is suitable for short-distance communication where the user may not always be directly in front of the microphone.
- **Flexible gooseneck design**: Allows users to adjust the mic position for optimal voice clarity.
- **Mute switch**: Provides control for the user to mute the mic when needed.

2.3.2 Role of the Microphone in Our Project

The ZAFFIRO M-DU02 is the **primary audio input** device in our secure communication system. It serves several essential functions:

- **Capturing Voice Messages**: When the user wants to send a voice message, the microphone captures the audio input in real time. This audio is then saved in .wav format using Linux tools like arecord.
- **Preprocessing the Audio**: The captured audio may be processed (e.g., noise filtering or normalization) to ensure clarity during playback on the receiver's device.
- **Encryption and Transmission**: After capturing and optionally preprocessing the voice, the audio file is **encrypted using symmetric or asymmetric encryption methods** before being sent over the network.
- **Integration with UI Events**: The user interface developed using LVGL triggers recording events when the microphone is active, and the captured audio is linked to a visual "voice message bubble."
- **Testing and Debugging**: The microphone was essential during the development phase for testing the audio pipeline, latency measurements, and adjusting encryption payload sizes.

2.3.3 Why We Chose the ZAFFIRO M-DU02

Several factors influenced our decision to use the ZAFFIRO M-DU02 microphone:

- **Plug-and-Play Compatibility with Raspberry Pi**
  Unlike other professional microphones requiring additional drivers or sound cards, the ZAFFIRO M-DU02 works out of the box with Raspberry Pi. Using standard Linux tools (arecord, ALSA), we could instantly record and access audio data.

- **Cost Efficiency**
  Being a budget-friendly microphone, the M-DU02 fits well within student project constraints. It offers acceptable sound quality without the high costs of studio-grade equipment.

- **Sufficient Audio Quality for Voice Encryption**
  We're not aiming for high-fidelity music recording. The clarity needed for voice recognition, transmission, and playback is met by the M-DU02. It records clean voice audio that is easy to encrypt and decode.

- **USB Power and Portability**
  Powered directly via USB, the microphone reduces the need for external power supplies. It also occupies a single USB port, leaving GPIO pins and other interfaces free for the touchscreen, speaker, and other modules.

- **Mechanical Flexibility**
  Its adjustable neck makes it more user-friendly, especially when used with a touchscreen interface. Users can simply bend it to face them, improving voice pickup.

- **Muting Feature**
  The hardware mute switch enhances user control over their privacy. It acts as an additional safeguard when transmitting sensitive voice data.

2.3.4 Conclusion

The ZAFFIRO USB Desktop Microphone M-DU02 has proven to be a practical and effective solution for capturing voice data in our secure communication system. Its ease of integration, affordability, and adequate performance make it an ideal choice for embedded projects involving the Raspberry Pi. As voice communication plays a central role in our project, selecting the right microphone was a strategic decision—one that directly impacts user experience, system reliability, and project success.

2.4 MINI USB-POWERED SPEAKER

2.4.1 Description of the Mini USB Speaker

This type of speaker is very common. It is usually used with laptops or desktop computers. It comes as a small stereo speaker with two parts: the **audio plug** and the **power plug**.

- The **audio plug** is a **3.5mm jack** (the same type used in headphones), and it connects to the **audio port on the Raspberry Pi**.
- The **USB plug** gives power to the speaker by connecting it to any **USB port on the Raspberry Pi**.

These speakers do not need any drivers or software. They work automatically when connected.

2.4.2 Features of the Speaker

Here are some important features that make this speaker a good choice for our project:

Table 2-7: Mini USB-speaker features.

Feature	Description
Power Source	USB-powered, works with 5V from Raspberry Pi
Audio Connection	3.5mm jack, connects to Raspberry Pi headphone port
Size	Small and portable, fits easily beside Raspberry Pi and touchscreen
Sound Quality	Clear enough for voice playback, suitable for short and simple messages
Easy Setup	Plug-and-play, no drivers or complex setup required
Volume Control	Some models come with a volume knob for easy sound adjustment

2.4.3 Why We Chose This Speaker

- **Simple to Use**
  The speaker is very easy to connect. We just plug the audio cable into the Raspberry Pi's headphone port and the USB cable into a USB port. This allows us to use it right away without any software installation.

- **Clear Voice Playback**
  Although the speaker is small, the sound is loud and clear enough to hear voice messages. Since our project focuses on talking and listening, we do not need very high-quality sound like music systems. This speaker provides just the right quality for hearing spoken words clearly.

- **Small and Portable**
  The speaker is small and fits well next to the Raspberry Pi and touchscreen. It does not take up much space, so it is good for projects that are designed to be used on a desk or small table.

- **Uses USB Power**
  The Raspberry Pi already has USB ports, so we do not need an external power supply for the speaker. It gets its power directly from the Pi, which makes the design cleaner and safer.

- **Works on Linux/Raspberry Pi**
  Because this speaker uses the standard 3.5mm and USB connection, it works perfectly with the Raspberry Pi's operating system (Raspberry Pi OS). It can be used with simple Linux commands like aplay to play sound files.

2.4.4 Conclusion

The **Mini USB-powered speaker** with a 3.5mm audio cable is the best choice for our project. It is simple, clear, and works perfectly with the Raspberry Pi. It helps us hear the encrypted voice messages after they are received and decrypted. The speaker completes the communication process, allowing users to both **send and receive** messages using a full audio setup.

Because it is small, easy to use, and reliable, this speaker fits well with the overall goal of the project: creating a safe, portable, and secure system for sending and receiving encrypted voice and text between Raspberry Pi devices.

2.5 TRANCEIVER (NRF24L01)

2.5.1 Introduction to nRF24L01

The nRF24L01 is a highly integrated, ultra-low power (ULP) 2.4 GHz RF transceiver IC. Its low cost, versatility, and ease of use have made it popular for short-range wireless communication in various embedded systems. It operates in the Industrial, Scientific, and Medical (ISM) radio band, which is a globally available and license-free frequency spectrum.

2.5.2 Key Features:

- **2.4 GHz ISM Band:** This frequency band (2.400 - 2.4835 GHz) is internationally recognized for industrial, scientific, and medical uses. Devices operating in this band generally do not require special licenses.
- **Data Rate:** Supports configurable air data rates of 250 kbps, 1 Mbps, and 2 Mbps.
- **Automatic Packet Handling:** Hardware-level support for packet handling.
- **Automatic Packet Retransmission:** Improves reliability by retransmitting packets if no acknowledgment (ACK) is received.
- **Automatic Acknowledgement (Auto-ACK):** Receiver automatically sends an ACK upon successful reception.
- **MultiCeiver (Enhanced ShockBurst):** Allows the module to listen for incoming transmissions on up to 6 different data pipes (addresses).
- **Low Power Consumption:** Designed for power efficiency, suitable for battery-powered applications.

2.5.3 Peer-to-Peer Communication Explained

In a true peer-to-peer (P2P) wireless setup, two nRF24L01 modules communicate directly without an intermediary. For bidirectional communication, each device needs two "communication pipes" configured:

- **A Sending Pipe (Writing Pipe):** An address that the device uses to transmit data to the other device.
- **A Receiving Pipe (Reading Pipe):** An address that the device listens on to receive data from the other device.

To achieve continuous two-way communication:

- Device A must know the address that Device B is listening on to send data to B.
- Device A must also listen on an address that Device B knows about to receive data from B.

Similarly for Device B:

- Device B must know the address that Device A is listening on to send data to A.
- Device B must also listen on an address that Device A knows about to receive data from A.

Define two distinct pipe addresses (e.g., address1 and address2).

```
radio.openWritingPipe(address2); //Sends data to Device B
radio.openReadingPipe(address1); //Receives data from Device B
```

Code Snippet 2-1: Device A's Configuration

```
radio.openWritingPipe(address1); // Sends data to Device A
radio.openReadingPipe(address2); // Receives data from Device
```

Code Snippet 2-2: Device B's Configuration

2.5.4 Hardware Connections (Raspberry Pi 3 Example)

Connecting the nRF24L01 module correctly is crucial.

Voltage Requirement (CRITICAL): The nRF24L01 operates at 3.3V. Supplying 5V directly to the VCC pin will almost certainly damage the module.

- The Raspberry Pi's GPIO pins operate at 3.3V, and it provides a 3.3V power rail.

- Capacitor Recommendation: Place a 10uF (or larger) electrolytic capacitor between the VCC and GND pins of the nRF24L01 module, as close to the module as possible, to help stabilize the power supply during transmission bursts.

The nRF24L01 communicates with the microcontroller using the **SPI (Serial Peripheral Interface)** protocol.

SPI Pins:

- SCK (Serial Clock): Clock signal generated by the master.
- MOSI (Master Out Slave In): Data sent from the master to the slave.
- MISO (Master In Slave Out): Data sent from the slave to the master.
- CSN (Chip Select Not): Active-low signal used by the master to select a specific slave device.
- CE (Chip Enable): Used to enable the radio's transmit or receive modes

Table 2-8: nRF24L01 Pinout

nRF24L01 Pin	Description
GND	Ground
VCC	3.3V Power Supply
CE	Chip Enable
CSN	Chip Select Not
SCK	Serial Clock
MOSI	Master Out Slave In
MISO	Master In Slave Out
IRQ	Interrupt Request (Optional)

Table 2-9: Connection to Raspberry Pi 3

nRF24L01 Pin	Raspberry Pi 3 GPIO Pin	Function
VCC	3.3V (Pin 1 or 17)	Power
GND	GND (Pin 6, 9, 14, 20, 25, 30, 34, 39)	Ground
CE	GPIO 22 (Pin 15)	Chip Enable
CSN	GPIO 8 (Pin 24)	Chip Select (SPI CE0)
SCK	GPIO 11 (Pin 23)	Serial Clock (SPI SCLK)
MOSI	GPIO 10 (Pin 19)	Master Out Slave In (SPI MOSI)
MISO	GPIO 9 (Pin 21)	Master In Slave Out (SPI MISO)
IRQ	*Optional* (e.g., GPIO 27 - Pin 13)	Interrupt Request

2.5.5 Software Setup (C Programming with RF24 Library)

We'll use the official RF24 C++ library by TMRh20. This library is designed to work with various SPI interfaces, including the Raspberry Pi's native SPI.

1. Enable SPI on your Raspberry Pi:

- Open a terminal and type: sudo raspi-config
- Navigate to Interface Options > SPI > Yes to enable it.
- Reboot your Raspberry Pi: sudo reboot

2. Install Git and build tools:

Code Snippet 2-3: Install Build Tools.

3. Clone the RF24 Library:

Code Snippet 2-4: clone RF24 repo.

4. Build and install the RF24 Library for Raspberry Pi

2.5.6 Communication Pipes and Addressing in Detail

The nRF24L01 uses "data pipes" to manage communication channels.

Each pipe has a unique address.

- Logical Pipes: The nRF24L01 supports one primary "receive" pipe (Pipe 0) and up to five additional "receive" pipes (Pipes 1-5).
- Addressing: A pipe address is a sequence of bytes, typically 5 bytes long.
  - Example: const byte address[6] = "00001";
- Uniqueness and Consistency: Each communication pipe should have a unique address.

Defining Addresses for Bidirectional Communication:

- const byte pipeA_RX_pipeB_TX[6] = "Node1"; // Device A listens on this, Device B writes to this
- const byte pipeB_RX_pipeA_TX[6] = "Node2"; // Device B listens on this, Device A writes to this

```
radio.openReadingPipe(1, pipeA_RX_pipeB_TX);  // Device A listens for data from Device B
radio.openWritingPipe(pipeB_RX_pipeA_TX);  // Device A sends data to Device B
```

Code Snippet 2-5: Device A Setup:

```
radio.openReadingPipe(1, pipeB_RX_pipeA_TX);  // Device B listens for data from Device A
radio.openWritingPipe(pipeA_RX_pipeB_TX);  // Device B sends data to Device A
```

Code Snippet 2-6: Device B Setup:

2.5.6.1 Core RF24 Library Functions Explained

- radio.begin(): Initializes the nRF24L01 module.
- radio.setDataRate(rate): Sets the air data rate. Options: RF24_250KBPS, RF24_1MBPS, or RF24_2MBPS.
- radio.setPALevel(level): Sets the Power Amplifier (PA) level. Options: RF24_PA_MIN, RF24_PA_LOW, RF24_PA_MED, or RF24_PA_HIGH.
- radio.setChannel(channel): Sets the RF channel (0-125).
- radio.openReadingPipe(pipe_number, address): Opens a data pipe for receiving.
- radio.openWritingPipe(address): Configures the module to transmit data to the specified address.
- radio.startListening(): Puts the radio into receive mode.
- radio.stopListening(): Puts the radio into transmit mode.
- radio.write(&data, size): Transmits data.
- radio.available(): Checks if there is data available to be read.
- radio.read(&buffer, size): Reads the received data.
- radio.printDetails(): Prints the radio's configuration registers to the Serial Monitor.

2.5.6.2 Advanced Concepts and Best Practices

- Payload Size: The nRF24L01's maximum payload size is 32 bytes.
- Acknowledgments (ACKs) and Retries: The RF24 library enables Auto-ACK and automatic retransmissions.
- Dynamic Payloads: Allows packets of varying sizes (up to 32 bytes) to be sent and received.
- radio.enableDynamicPayloads();
- radio.getDynamicPayloadSize() to determine the exact size of the received packet.
- ACK Payloads: Allows the receiver to send a small payload back to the sender along with the acknowledgment.
- Interrupts (IRQ Pin): Use the IRQ pin for more efficient (non-blocking) operation.

2.5.7 Troubleshooting Common Issues

1. **"Radio hardware not responding! Check connections." (from radio.begin()):**

    - Wiring: Double-check all SPI connections.
    - Power Supply: Confirm your nRF24L01 is receiving a stable 3.3V, NOT 5V.
    - Capacitor: A 10uF (or larger) capacitor across VCC and GND.

2. **"Message failed to send (no acknowledgment)." or No Data Received:**

    - Distance/Obstacles: Too far apart, or too many obstacles.
    - Mismatched Parameters:
    - Addresses: Ensure writing_pipe on the sender matches a reading_pipe on the receiver.
    - Channel: Both modules must be on the same RF channel (radio.setChannel()).
    - Data Rate: Both modules must use the same data rate (radio.setDataRate()).
    - PA Level: Ensure power levels are adequate for the distance.
    - Interference: Other 2.4 GHz devices can cause interference.
    - startListening() / stopListening(): Ensure these are correctly used.
    - Code Logic: Is the receiving loop actually checking radio.available() and calling radio.read()?

3. **One-way communication only:**

    - Incorrect openReadingPipe and openWritingPipe assignments.

Chapter Three

3 COMMUNICATION AND TCP

3.1 SOCKET PROGRAMMING IN C

3.1.1 Introduction Socket programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server. Socket programming is widely used in instant messaging applications, binary streaming, and document collaborations, online streaming platforms

3.1.2 Components of Socket Programming

Sockets
Sockets are one of the core components used by the program to access the network to communicate with other processes/nodes over the network. It is simply a combination of an IP address and a port number that acts as an endpoint for Communication.

Example
192.168.1.1:8080, where the two parts separated by the colon represent the IP address (192.168.1.1) and the port number (8080).

Socket Types:

- TCP Socket (Stream Socket): Provides reliable, connection-based communication (i.e., TCP protocol).

- UDP Socket (Datagram Socket): Provides connectionless communication, faster but unreliable (i.e., UDP protocol).

3.2 INTRODUCTION TO TCP

TCP (Transmission Control Protocol) is one of the main protocols in the Internet Protocol Suite, alongside IP (Internet Protocol).

It provides reliable, ordered, and error-checked delivery of data between applications, running on hosts communicating via a network Developed in the 1970s by Vint Cerf and Bob Kahn, TCP is widely used in applications such as web browsing, email, file transfers, and streaming, where data integrity is critical and Our project implements a TCP-based file transfer system to enable reliable data exchange between devices.

3.3 PROPERTIES OF TCP

TCP (Transmission Control Protocol) has several key properties that ensure reliable data transfer:

1. **Reliable Data Transfer:**
    1. TCP ensures that data packets are delivered without loss or corruption. If a packet is lost, TCP requests retransmission.

    2. Each packet is acknowledged by the receiver. For example, if the sender sends packets P1, P2, and P3, the receiver sends acknowledgements to confirm successful receipt.

2. **Organizing Data:**
    1. TCP reassembles packets into the correct order.

    2. This ensures that the data is reconstructed into a meaningful message at the receiving end.

3. **Error Checking:**
    1. TCP checks data for errors and requests replacements for corrupted packets, enhancing reliability.

2. **Connection-Oriented:**
    1. TCP establishes a connection between sender and receiver before transmitting data.

    2. This connection is established through a **three-way handshake** and terminated via a **four-way handshake**.

3.4 HOW DOES TCP WORK

Sending data over a TCP connection involves three primary phases:

1. Connection Establishment (Three-way handshake)
2. Data Transfer
3. Connection Termination (Four-way handshake)

Three-Way Handshake

A method of establishing a connection in TCP This involves the following steps:

1. **Synchronize (SYN):**

   - The client sends a TCP segment with the SYN flag set, indicating a request to establish a connection.

   - This segment includes the client's initial sequence number (Seq = X) . Client: "Hey server, I want to maintain a connection."

2. **Synchronize-Acknowledge (SYN-ACK):**

   - The server responds with a TCP segment with both SYN and ACK flags set.

   - The ACK flag acknowledges the client's SYN (Ack = X + 1).

   - The SYN flag indicates the server's willingness to establish a connection, along with its initial sequence number (Seq = Y).

   - Server: "Yes, I received your connection message, and I also want to make the connection."

3. **Acknowledge (ACK):**

   - The client sends an ACK packet to the server, acknowledging the server's SYN (Ack = Y + 1).

   - The connection is now established, and data transfer can begin. Client: "Let's connect."

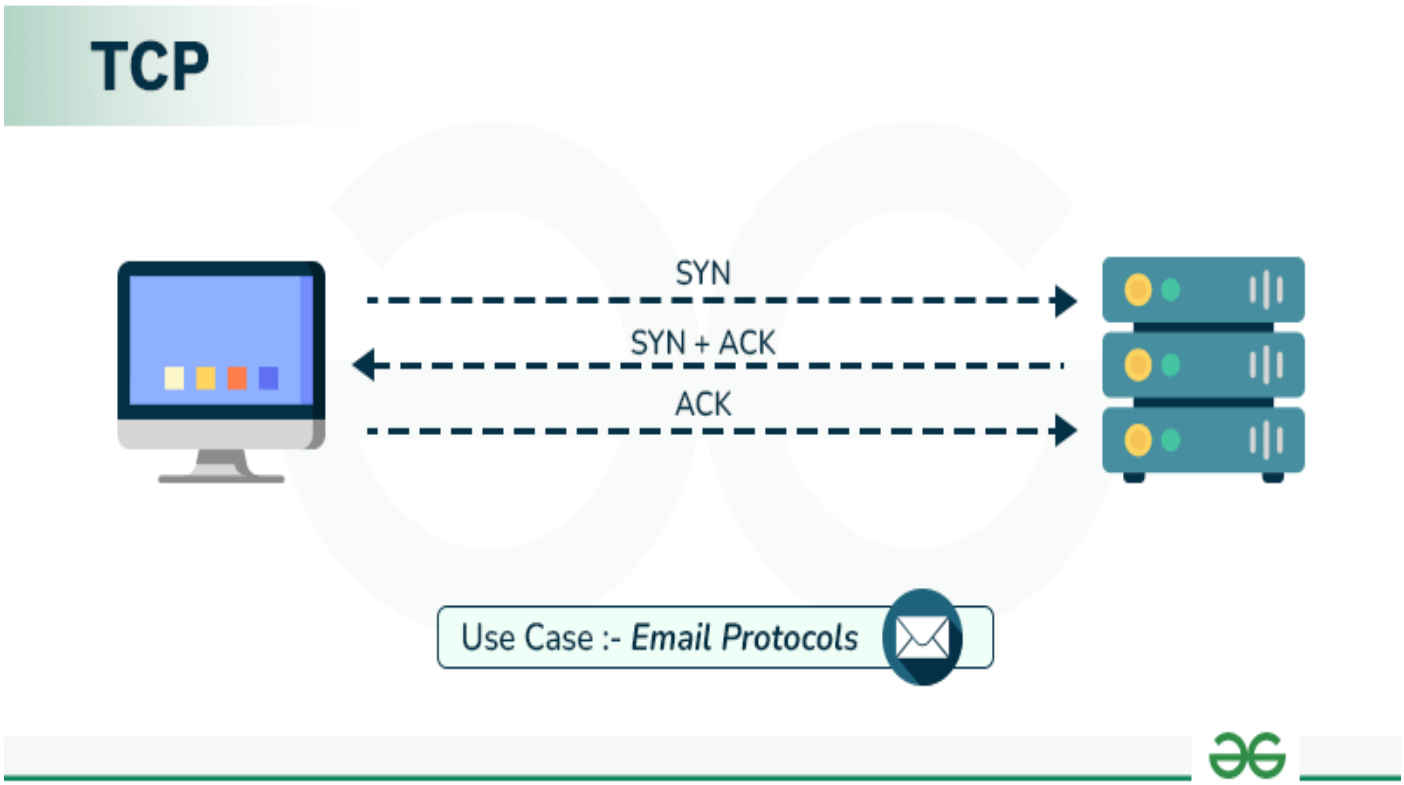


Figure 3-1: Illustrate Three Way Handshake.

Data Transfer

Once the three-way handshake is complete, data transfer occurs as follows:

1. Segmenting Data:
    - The sender divides the data into TCP segments, each with a sequence number.
    - For example, 1000 bytes of data might be split into multiple segments.

2. **Acknowledgment:**
    - The receiver sends an ACK for each received segment.
    - If a segment is lost or an error occurs, the sender retransmits it.

Four-Way Handshake

A method of terminating a connection in TCP, when data transfer is complete, TCP terminates the connection using a four-way handshake:

1. **FIN:**
    - The client sends a FIN packet to the server, indicating it wants to close the connection.
2. **ACK**:
    - The server acknowledges the FIN with an ACK packet.
3. **FIN**:
    - The server sends its own FIN packet to the client, indicating it is ready to close the connection.
4. **ACK**:
    - The client acknowledges the server's FIN with an ACK packet.

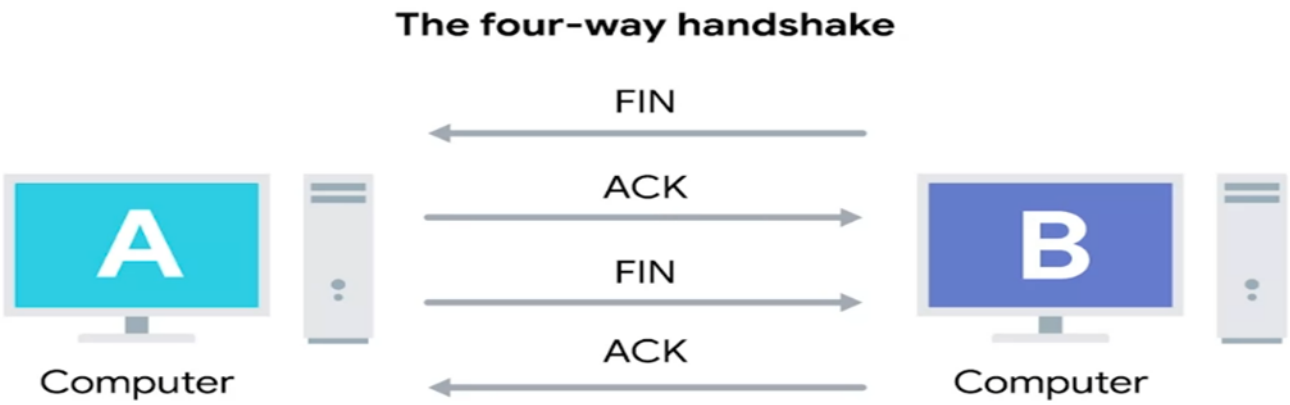


Figure 3-2: Illustrate Four Way Handshake.

3.5 UNDERSTANDING TCP THROUGH VISUALIZATION

Table 3-1: TCP flags Table.

Term	Description
Seq	Sequence Number
Ack	Acknowledgement Number
SYN	SYN flag, indicates a synchronize packet ACK
	ACK flag, indicates an acknowledgement packet FIN
	FIN flag, indicates a finish packet
Client's Initial Sequence Number	A random number, e.g., 1000
Server's Initial Sequence Number	A random number, e.g., 2000

3.5.1.1 Connection Establishment Example

1. **Client Sends SYN to Server:**
   - The client sends a SYN packet with a sequence number of 1000.

2. **Server Sends SYN-ACK to Client:**
   - The server acknowledges the client's sequence number by adding 1 (1000 + 1 = 1001) and sends its sequence number 2000.

3. **Client Sends ACK to Server:**
   - The client acknowledges the server's sequence number by adding 1 (2000 + 1 = 2001) and sends it to the server.

3.5.1.2 Data Transfer Example

Suppose the client sends "Hello World" (4000 bytes), divided into two segments:

- Segment 1: "Hello" (2000 bytes)
- Segment 2: "World" (2000 bytes)

4. **Segment 1 – Client → Server:**
   - The client sends the first segment with a sequence number of 1001 (initial sequence number + 1).

5. **ACK for Segment 1 – Server → Client:**
   - The server receives the first 2000 bytes and sends an ACK. The server's ACK number is 3001 (1001 + 2000).

6. **Segment 2 – Client → Server:**
   - The client sends the second segment with a sequence number of 3001.

7. **ACK for Segment 2 – Server → Client:**
   - The server receives the second 2000 bytes and sends an ACK.
   - The server's ACK number is 5001 (3001 + 2000), indicating all 4000 bytes have been received.

3.5.1.3 Connection Termination Example

1. **FIN – Client → Server:**
   - The client sends a FIN packet with a sequence number of 5001.

2. **ACK for FIN – Server → Client:**
   - The server acknowledges the client's FIN packet with an ACK, setting the ACK number to 5002 (5001 + 1).

3. **FIN – Server → Client:**
   - The server sends its own FIN packet with a sequence number of 5001.

4. **ACK for Server's FIN – Client → Server:**
   - The client acknowledges the server's FIN with an ACK, setting the ACK number to 5002 (5001 + 1).

For a true "peer-to-peer" bidirectional TCP connection, one Raspberry Pi will act as the server (listening for incoming connections) and the other as the client (initiating the connection). Once the connection is established, both can send and receive data simultaneously.

3.5.2 Client-Server Model

The client-server model refers to the architecture used in socket programming, where a client and a server to interact with each other to exchange information or services. This architecture allows client to send service requests and the server to process and send response to those service requests.State Diagram for Server and Client ModelCreating a Server-Side Process.

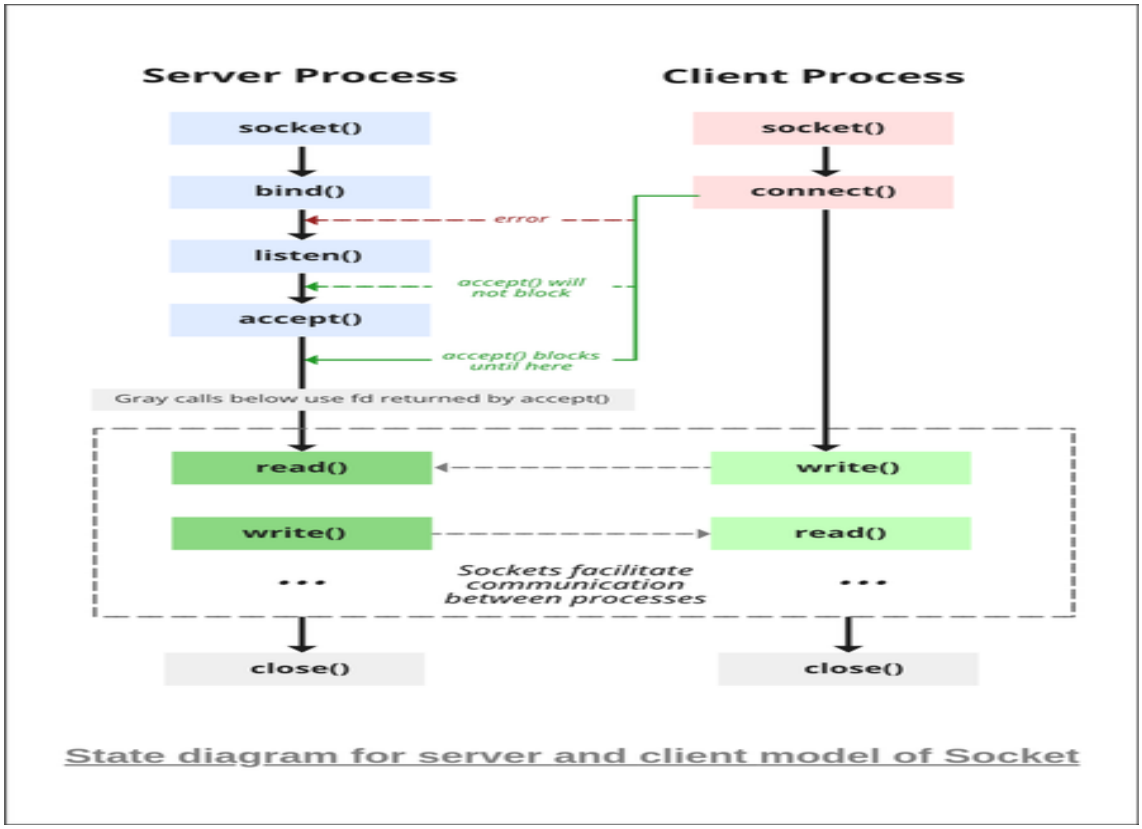
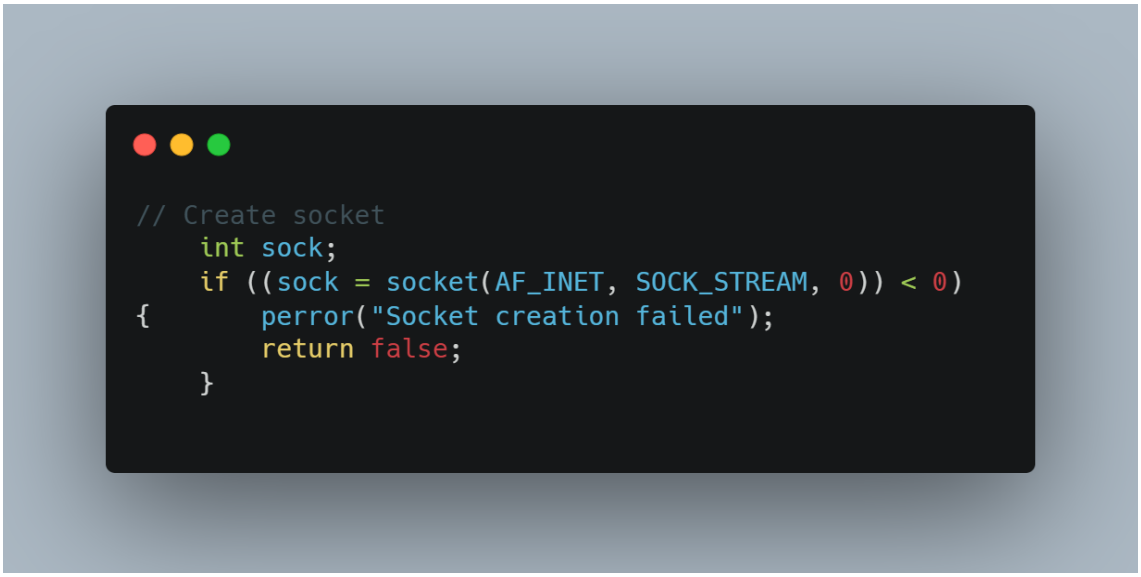


Figure 3-3: Client Server Connection establish.

The server is created using the following steps:

1. Socket Creation with socket () function



Code Snippet 3-1: Create Socket.

1. domain (Communication Domain)

Table 3-2: Specifies the address family used for communication

Type	Description
SOCK_STREAM	TCP: Connection-oriented, reliable, byte-stream (uses sequencing/ACKs)
SOCK_DGRAM	UDP: Connectionless, unreliable, fixed-size datagrams
SOCK_RAW	Raw network protocol access (requires root privileges)
Type	Description

2. type (Socket Type)

Table 3-3: Determines the communication semantics

Type	Description
SOCK_STREAM	TCP: Connection-oriented, reliable, byte-stream (uses sequencing/ACKs)
SOCK_DGRAM	UDP: Connectionless, unreliable, fixed-size datagrams
SOCK_RAW	Raw network protocol access (requires root privileges)

Key Differences:

- TCP (SOCK_STREAM): Automatically handles retransmission, ordering, and congestion control.
- UDP (SOCK_DGRAM): Faster but requires manual error handling.

3. Protocol

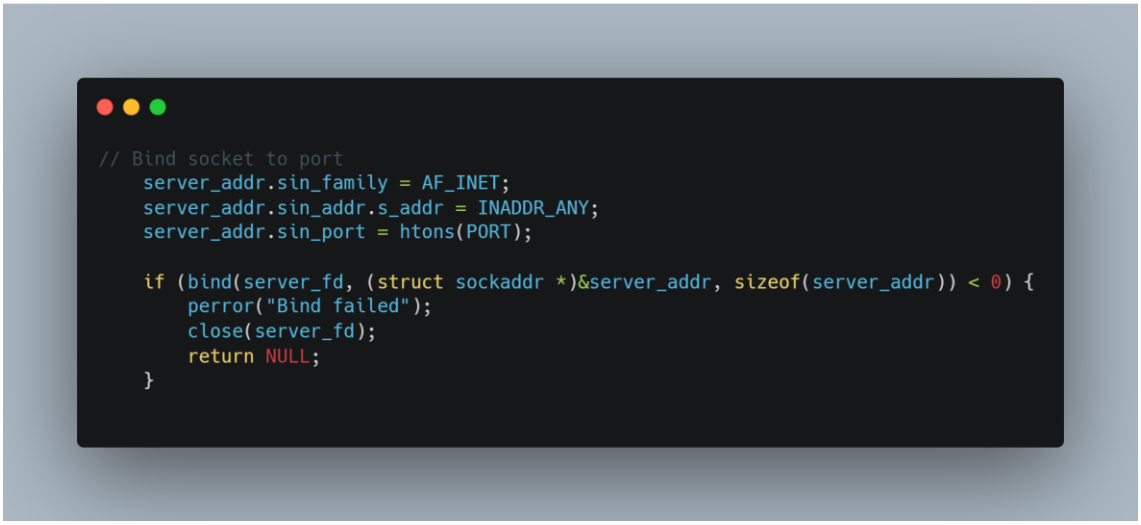
Specifies a particular protocol to use with the socket. Typically set to 0 for automatic selection based on domain and type:

- SOCK_STREAM + AF_INET → Automatically chooses TCP
- SOCK_DGRAM + AF_INET → Automatically chooses UDP

4. Return Value (sockfd)

- On success: Returns a socket descriptor (non-negative integer, similar to a file handle).
- On failure: Returns -1, and errno is set to indicate the error.

2. Bind



```
// Bind socket to port
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(server_fd);
        return NULL;
    }
```

Code Snippet 3-2: Bind Socket.

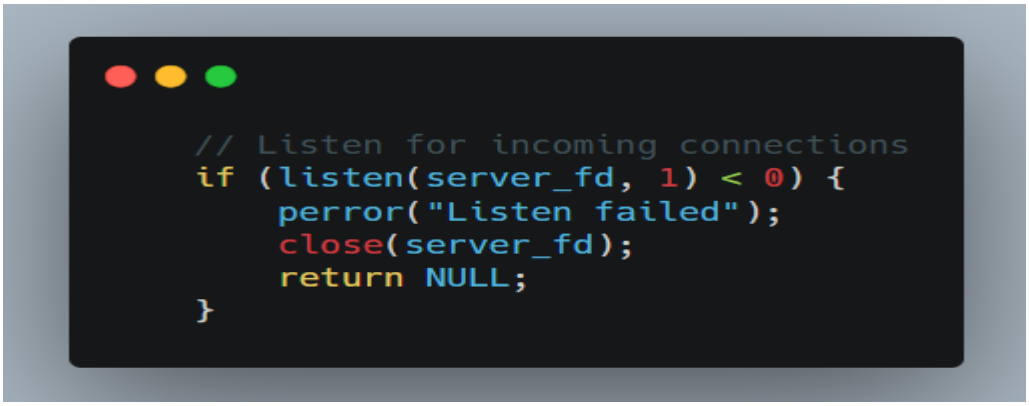
After the creation of the socket, the bind () function binds the socket to the address and port number specified in addr(custom data structure)

- Servers: To listen on a specific port (e.g., HTTP on port 80).
- Clients: Rarely used (the OS assigns a random port by default).

Table 3-4: Bind Parameters.

Parameter	Type	Description
sockfd	int	Socket descriptor returned by socket().
addr	struct sockaddr*	Pointer to address structure (IPv4: sockaddr_in, IPv6: sockaddr_in6).
addrlen	socklen_t	Size of the address structure (e.g., sizeof(struct sockaddr_in)).

3. Listen for incoming connections



```
// Listen for incoming connections
if (listen(server_fd, 1) < 0) {
    perror("Listen failed");
    close(server_fd);
    return NULL;
}
```

Code Snippet 3-3: listen to incoming connection.

4. Connection Acceptance (accept())

The server enters its final preparation phase by implementing the accept() system call, which completes the TCP three-way handshake and establishes dedicated communication channels with clients.

```
// Accept a client connection
    if ((client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &client_len)) < 0)
{       perror("Accept failed");
        close(server_fd);
        return NULL;
    }
```

Code Snippet 3-4: Accept connection.

1. **Functional Overview**
   - Extracts the first pending connection from the listen queue
   - Creates a new socket dedicated to the client connection
   - Returns a new file descriptor for data exchange
   - Blocks indefinitely until a connection arrives (by default)

   Parameters:
   - sockfd: socket file descriptor returned by socket() and bind().
   - addr: pointer to a struct sockaddr that will be hold the client's IP address and port number.

   - addrlen: pointer to a variable that specifies the length of the address structure.

5. Sending Data/file

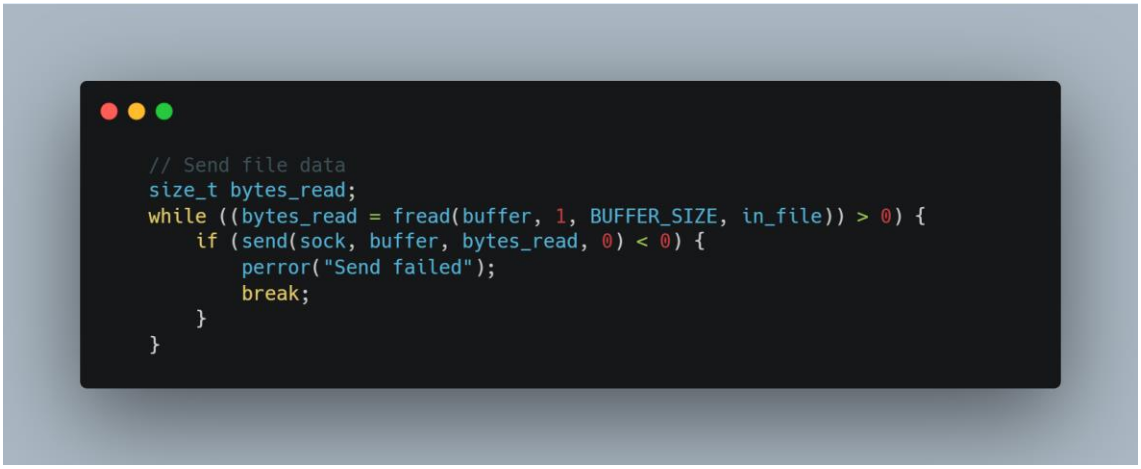
For sending I will send file name first and then the data which will contain in a file name or I can send string only without but it in file

1. Send file name

```
// Then send the filename
if (send(sock, filename, strlen(filename), 0) < 0) {
    perror("Failed to send filename");
    close(sock);
    fclose(in_file);
    return false;
}
```

Code Snippet 3-5: Send File Name.

2. send file data



```
// Send file data
size_t bytes_read;
while ((bytes_read = fread(buffer, 1, BUFFER_SIZE, in_file)) > 0) {
    if (send(sock, buffer, bytes_read, 0) < 0) {
        perror("Send failed");
        break;
    }
}
```

Code Snippet 3-6: Send File Data.

6. Receiving Data/File

For receiving I will receive file name first and then the data which will contain in a file name or I can receive string only without but it in file.

1. receive file name

```
// Then receive the filename
    int total_received = 0;
    while (total_received < filename_len) {
        int bytes = recv(client_fd, filename + total_received, filename_len - total_received, 0);
        if (bytes <= 0) {
            perror("Failed to receive complete filename");
            return false;
        }
        total_received += bytes;
    }
```

Code Snippet 3-7: Receiving File Name.

2. Receive file data

```
// Receive file data
    ssize_t bytes_received;
    while ((bytes_received = recv(client_fd, buffer, BUFFER_SIZE, 0)) > 0) {
        if (fwrite(buffer, 1, bytes_received, out_file) != bytes_received) {
            perror("Write failed");
            break;
        }
    }
```

Code Snippet 3-8: Receiving file data.

7. Close

After the exchange of information is complete, the server closes the socket using the close () function and releases the system resources.

Code Snippet 3-9: Close connection.

Summary for Pi-Server (Listener):

1. Create Socket: socket()
2. Bind: Assigns a local IP address and port number to the socket bind()
3. Listen: Puts the socket into a listening state, waiting for incoming connections listen()
4. Accept: Blocks until a client connects and then creates a new connected socket for that client accept()
5. Communicate: Use the new connected socket to send() and recv() data.
6. Close: Closes the sockets close() when done.

Creating Client-Side Process

1. Socket connection

- This step involves the creation of the socket which is done in the same way as that of server's socket creation

2. Connect

- The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

3. Send/Recieve

- In this step the client can send or receive data from the server which is done using the send() and recieve() functions similar to how the server sends/recieves data from the client.

4. Close

- Once the exchange of information is complete, the client also needs to close the created socket and releases the system resources using the close() function in the same way as the server does.

Summary for Pi-Client (Initiator):

1. Create Socket: socket()
2. Connect: Initiates a connection to the Pi-Server's IP address and port connect()
3. Communicate: Use the same socket to send() and recv() data.
4. Close: Closes the socket close() when done.

3.6 HARDWARE SETUP (RASPBERRY PI 3)

For TCP communication, your Raspberry Pis simply need network connectivity. This can be via:

- **Ethernet Cables**: Direct connection between the two Pis, or both connected to the same router/switch. This is often the most reliable for initial testing.

- **Wi-Fi:** Both Pis connected to the same Wi-Fi network.

No special external hardware like nRF24L01 modules is needed for TCP sockets.

Network Configuration:

1. **Identify IP Addresses:** You'll need the IP addresses of both your Raspberry Pis.

    - Open a terminal on each Pi and type: hostname -I (that's a capital 'i').

    - Note down the IP address for each Pi. For example, Pi-Server might be 192.168.1.100 and Pi-Client might be 192.168.1.101.

2. **Choose a Port Number:** Select an unused port number (e.g., 8080, 12345, 5000). Avoid well-known ports (0-1023) as they are reserved for standard services.

    - Crucially: Both the client and server must agree on the same port number.

3.7 SOFTWARE SETUP

You'll need a C compiler (gcc), which is usually pre-installed on Raspberry Pi OS.

Basic C Socket Headers:
The core C socket functions are provided by standard system libraries. You'll primarily use headers like:

- <stdio.h>: Standard input/output.
- <stdlib.h>: Standard library functions (e.g., exit).
- <string.h>: String manipulation functions (e.g., strlen, strcpy).
- <sys/socket.h>: Core socket definitions.
- <netinet/in.h>: Internet address family (for sockaddr_in).
- <arpa/inet.h>: Functions for IP address conversion (e.g., inet_pton).
- <unistd.h>: POSIX operating system API (for close, read, write).

Table 3-5: Summary of Key Socket Concepts and Functions Explained

Concept/Function	Description
socket(AF_INET, SOCK_STREAM, 0)	Creates a new socket. **AF_INET** specifies IPv4, **SOCK_STREAM** specifies TCP (a stream socket), and **0** selects the default protocol for the given family and type.
sockaddr_in Structure	Holds address information. Includes address family (**sin_family**), port number (**sin_port**), IP address (**sin_addr**), and padding.
htons(PORT)	"Host to Network Short." Converts the port number from host byte order to network byte order (required for network communication).
INADDR_ANY	For the server, this means the socket will listen on all available network interfaces.
inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr)	Converts an IP address from text (e.g., "192.168.1.100") to its binary representation, used by **sockaddr_in**.
bind(server_fd, (struct sockaddr *)&address, sizeof(address))	Associates the created socket (**server_fd**) with a specific local IP address and port number. Only done by the server.
listen(server_fd, 3)	Puts the server socket into a passive listening state. **3** is the maximum number of pending connections in the queue (backlog).
accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)	Blocks the server until a client tries to connect. When a connection comes in, **accept** creates a new socket for that specific client connection and returns its file descriptor (**new_socket**). The original **server_fd** continues to listen for new clients.
connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))	Attempts to establish a connection to the remote server specified by **serv_addr**. Only done by the client.
send(socket_fd, buffer, length, flags)	Sends data through the socket.
recv(socket_fd, buffer, length, flags)	Receives data from the socket.
close(socket_fd)	Closes a socket connection.

3.8 DISADVANTAGES OF TCP

While TCP offers reliability, it has drawbacks, especially when speed and low latency are crucial.

1. Overhead and Complexity:

- TCP's reliability mechanisms add complexity and overhead. It maintains stateful information (sequence numbers, ACK numbers) and handles congestion control, error checking, and retransmissions.

2. Latency Due to Reliability Mechanisms:

- TCP waits for ACKs for each packet. Retransmission of lost packets introduces latency, especially in congested networks.

3. No Support for Broadcast or Multicast:

- TCP supports only unicast communication, requiring separate connections for each recipient in applications like video streaming.

4. Head-of-Line Blocking:

- Data delivery is in order, so if one packet is delayed, subsequent packets must wait, slowing down the entire transmission.

5. Connection-Oriented Nature:

- The three-way handshake introduces overhead, especially for applications sending small amounts of data quickly (e.g., IoT sensors) .

Chapter four

4 ENCRYPTION AND SECURITY PROTOCOLS

Secure communication is a cornerstone of modern digital systems, particularly in networks where sensitive data is transmitted between nodes such as Raspberry Pi (RPI) devices. In any project involving the exchange of information, encryption serves as the primary mechanism to safeguard both text and voice messages from eavesdropping or tampering. However, a truly robust security posture extends beyond the mere application of encryption; it requires a holistic architecture grounded in established cybersecurity principles, a clear understanding of the threat landscape, and a pragmatic approach to balancing security with performance. For a network of RPI nodes, which may operate in diverse and potentially untrusted environments, establishing a secure communication channel is not an optional feature but a fundamental requirement for operational integrity and data protection.

4.1 DEFINING SECURITY GOALS: THE CIA TRIAD AND BEYOND

4.1.1 The Imperative for Secure Communication

The development of any secure system must begin with a formal definition of its security objectives. The most widely accepted model for this is the **CIA Triad**, a foundational framework in cybersecurity policy that consists of three core principles: Confidentiality, Integrity, and Availability. These principles provide a comprehensive lens through which to evaluate threats and design effective countermeasures.

- **Confidentiality**: This principle is concerned with preventing the unauthorized disclosure of information. It ensures that data is accessible only to authorized parties. In the context of an RPI network exchanging message, confidentiality guarantees that even if a malicious actor intercepts a data packet, its contents whether a text message or a segment of a voice remain unintelligible and secret. This is the primary goal achieved through the process of encryption, which transforms readable data into a scrambled, secret code.

- **Integrity**: This principle ensures the accuracy, consistency, and trustworthiness of data throughout its entire lifecycle. For this project, integrity means that a message received by one RPI node is identical to the message sent by another, with no unauthorized modifications, additions, or deletions having occurred in transit. An attacker who tampers with a message, perhaps altering a command or changing financial details, violates its integrity. This is typically enforced using cryptographic mechanisms like hash functions and message authentication codes, which act as a digital seal to detect any changes.

- **Availability**: This principle guarantees that information and communication systems are operational and reliably accessible to authorized users when needed. For a network of RPI nodes, this means the communication channel must remain functional and performant. A denial-of-service (DoS) attack that crashes a node or a security measure so computationally intensive that it renders real-time voice communication unusable would both constitute a failure of availability.

While the CIA Triad is foundational, modern security architectures often extend this model to include two additional principles that are critical for building trust between network nodes:

- **Authenticity**: This verifies that the origin of data is legitimate. The receiving RPI node must have a high degree of certainty that a message purporting to be from "Node A" was, in fact, sent by Node A and not an impostor. Digital certificates and signatures are key technologies for establishing authenticity.

- **Non-Repudiation**: This principle provides accountability by ensuring that a sender cannot later deny having sent a message. It creates a verifiable record of transactions, which is crucial in systems where actions must be auditable.

These principles are not merely theoretical constructs; they are deeply embedded in modern regulatory and compliance frameworks such as the General Data Protection Regulation (GDPR) and ISO 27001, which explicitly reference confidentiality, integrity, and availability as required security measures. A critical aspect of system design is recognizing the inherent tension between these principles. For example, implementing a very strong encryption algorithm to maximize confidentiality might introduce significant processing delays (latency) on a resource-constrained RPI. This added

latency could degrade a real-time voice call to the point of being unusable, thereby compromising availability. Therefore, the central design challenge is not simply to apply security, but to engineer a balanced system that provides

4.1.2 The Threat Landscape: Attack Vectors Against RPI Networks

To implement effective security, one must first understand the specific threats the system faces. A network of RPI nodes, potentially communicating over both wired and wireless links, is susceptible to a range of well-understood cyberattacks. The security architecture must be designed to directly counter these vectors.

- **Eavesdropping via Packet Sniffing**: One of the most straightforward attacks against an unsecured network is eavesdropping. An attacker can use readily available software tools like Wireshark or tcpdump to place a network interface into "promiscuous mode," allowing it to capture all data packets traversing the local network segment, not just those addressed to it. If communication between RPI nodes is unencrypted, the content of these packets—including text messages, VoIP conversations, passwords, and network configuration data—can be read in plaintext. This passive attack directly compromises the **confidentiality** of all transmitted data.

- **Tampering and Impersonation via Man-in-the-Middle (MITM) Attacks**: A more active and dangerous threat is the Man-in-the-Middle (MITM) attack. In this scenario, an attacker positions themselves between two communicating RPI nodes, intercepting all traffic between them. The attack typically unfolds in two phases: interception and decryption/manipulation. The attacker might impersonate each node to the other, relaying messages while secretly reading and potentially altering them before forwarding them. This allows the attacker not only to eavesdrop (violating **confidentiality**) but also to tamper with the data in transit (violating **integrity**). Common techniques to initiate a MITM attack include ARP spoofing, where an attacker tricks devices on a local network into sending traffic to the attacker's machine instead of the legitimate gateway, and DNS spoofing, which redirects traffic to a malicious server. Creating a rogue Wi-Fi access point with a

convincing name is another common method to lure devices into connecting through an attacker-controlled channel.

- **Data and Command Injection Attacks**: Should an attacker succeed in intercepting a communication channel, they may attempt to go beyond passive eavesdropping or simple data alteration by injecting malicious payloads into the data stream. This attack exploits an application's failure to properly validate or sanitize its input data. For instance, if one RPI is sending control commands to another, an attacker could inject their own commands to cause unintended and potentially harmful actions. Similarly, an attacker could inject malicious scripts into a text message that, when processed by the receiving application, could lead to a system compromise. This form of attack is a direct assault on the **integrity** and can lead to severe consequences, including denial of service or complete system takeover.

4.2 SYMMETRIC VS. ASYMMETRIC ENCRYPTION

Two fundamental encryption paradigms are employed in this project:

4.2.1 AES (Advanced Encryption Standard) – Symmetric Encryption

How it works: AES uses a single shared key for both encryption and decryption. It is a block cipher, meaning it processes data in fixed-size blocks of 128 bits (16 bytes). These 16 bytes are arranged in a 4x4 matrix known as the "state." The algorithm then transforms this state over multiple rounds of processing. The number of rounds depends on the key size: 10 rounds for a 128-bit key, 12 for a 192-bit key, and 14 for a 256-bit key.

Each round consists of four distinct stages (except the final round, which omits the MixColumns step):

- **SubBytes:** This is a non-linear substitution step where each byte of the state is replaced with another according to a fixed lookup table called an S-box. This step provides confusion, obscuring the relationship between the key and the ciphertext. The S-box itself is generated from the multiplicative inverse over the finite field GF(2^8) combined with an affine transformation.

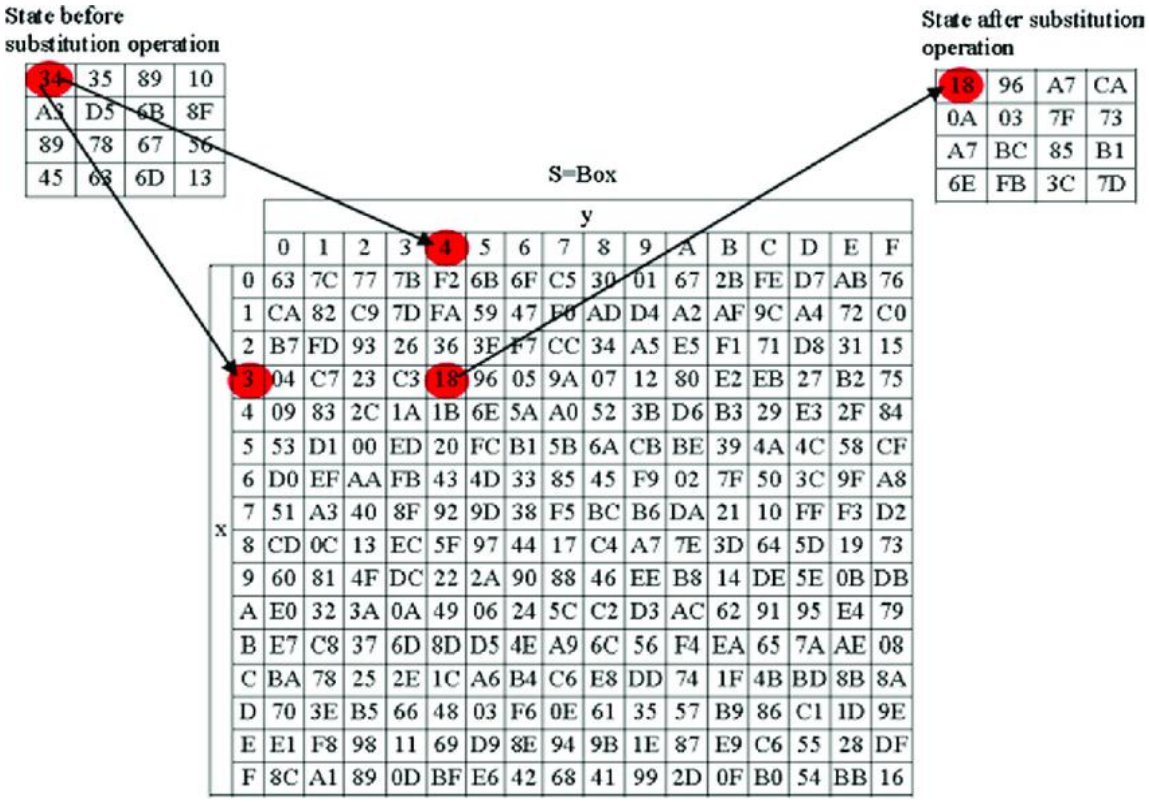


Figure 4-1: S-box matrix with explanation to SubBytes

- **ShiftRows:** This step provides diffusion by permuting the data. The rows of the 4x4 state matrix are cyclically shifted to the left by a specific offset. The first row is not shifted, the second row is shifted by one byte, the third by two bytes, and the fourth by three bytes.

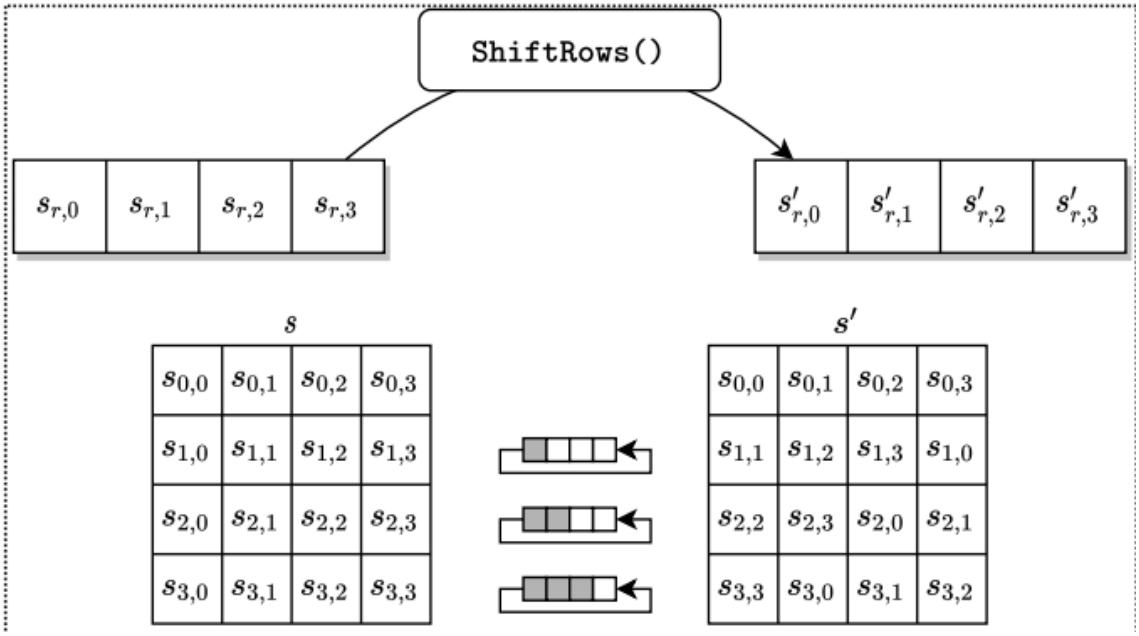


Figure 4-2: Illustration of SHIFTROWS

- **MixColumns:** To further enhance diffusion, this step operates on each column of the state individually. Each column is treated as a polynomial and multiplied by a fixed matrix in GF(2^8). The transformation for a single column can be represented as:

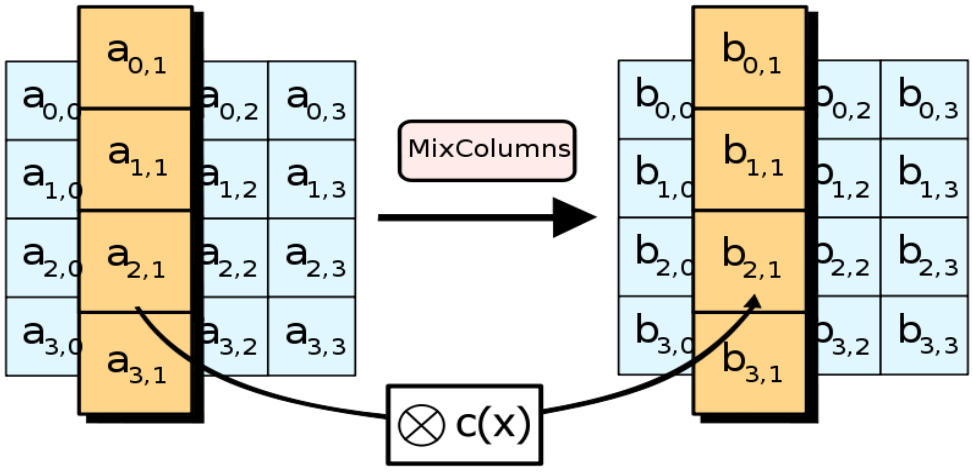


Figure 4-3: Illustration of MIXColumns

- **AddRoundKey:** In this final stage of the round, the state is combined with a unique round key using a simple bitwise XOR operation. The round keys are derived from the original cipher key via a process called the Key Expansion schedule.

Strengths:
- Fast processing, ideal for encrypting large data volumes (e.g., voice messages).
- Standardized by the U.S. National Institute of Standards and Technology (NIST) and widely adopted globally.

Weakness: Securely distributing the single shared key to all parties is a significant challenge.

4.2.2 RSA (Rivest-Shamir-Adleman) – Asymmetric Encryption

How it works: RSA uses a pair of mathematically linked keys: a public key (shared openly) to encrypt data and a private key (kept secret) to decrypt it. Its security relies on the principle that it is computationally easy to multiply two large prime numbers, but extremely difficult to factor their product back into the original primes.

The process involves three main stages:
Key Generation:
- Choose two distinct, large random prime numbers, p and q.
- Calculate their product to form the modulus:
$$n = p * q \quad (1)$$
- Calculate Euler's totient function for
$$\phi(n) = (p-1) * (q-1) \quad (2)$$
This function counts the positive integers up to n that are relatively prime to n.
- Choose a public exponent.
$$1 < e < \phi(n) \quad (3)$$
Where e is coprime to $\phi(n)$
- Compute the private exponent d as the modular multiplicative inverse of $e \% n$
- . This means finding d such that.
$$d \times e \equiv 1 \% \phi(n) \quad (4)$$

The **Public Key** is the pair (e, n) and the **Private Key** is (d, n).

Encryption: To encrypt a message M (represented as a number), the sender uses the recipient's public key with the following formula:

$$C = (M * e)\%n \quad (5)$$

Where C is the resulting ciphertext.

Decryption: The recipient uses their private key (d, n) to decrypt the ciphertext C and recover the original message M:

$$M = (C * D)\ \%\ n \quad (6)$$

Strengths:

- Solves the key distribution problem, as the public key can be shared without compromising security.
- Enables the creation of digital signatures, which are used to verify the authenticity and integrity of a message.

Weakness: The mathematical operations are far more complex and slower than symmetric encryption, making it unsuitable for encrypting large volumes of data.

4.2.3 Hybrid Approach

This project combines AES and RSA to leverage their respective strengths:

- RSA is used to securely exchange a temporary, one-time symmetric key (a "session key") for AES.
- AES then uses this fast and efficient session key to encrypt the actual message payloads (text/voice) for the duration of the communication.

4.3 SYSTEM SECURITY REQUIREMENTS

A secure communication system must be architected to address both passive and active threats. This section defines the threat model against which the Raspberry Pi network is hardened, establishes the formal security goals based on industry-standard principles, and outlines the specific cryptographic implementations required to meet those goals.

4.3.1 Threat Model

The system is designed to operate under the assumption that the underlying network is not secure. Adversaries may have the ability to monitor, intercept, and inject network traffic. The following adversarial scenarios are considered:

1. **Eavesdropping (Passive Attack)**

    - **Description:** An attacker on the same network can use packet sniffing software (e.g., Wireshark, tcpdump) to intercept data packets transmitted between the Raspberry Pi nodes. If communication is unencrypted, the attacker can read the content of text and voice messages in plaintext, directly compromising confidentiality.

    - **Mitigation:** All message payloads are encrypted using the AES-256 algorithm. This ensures that even if an attacker intercepts the data, its content remains unintelligible ciphertext, thereby preserving confidentiality.

2. **Man-in-the-Middle (MITM) Attack (Active Attack)**

    - **Description:** A more advanced adversary can position themselves between two communicating nodes, intercepting all traffic. The attacker can then read, modify, or inject new messages into the communication stream without the legitimate nodes' knowledge. This can be achieved through techniques like ARP spoofing or by setting up a rogue Wi-Fi access point.

    - **Mitigation:** The primary defense against MITM attacks is strong, mutual authentication. This system uses RSA-based digital signatures embedded within public-key certificates. Before any communication begins, each node verifies the other's certificate, confirming its identity and preventing an attacker from successfully impersonating a legitimate node.

3. **Replay Attack**

    - **Description:** An attacker captures a valid, encrypted message (e.g., a command to unlock a door) and retransmits it later to trick the system into performing the action again. Since the message itself is valid and properly encrypted, the system might accept it as a legitimate new command.

- **Mitigation:** To prevent this, each message is protected against replay. This is achieved by including a **timestamp** to ensure the message is only valid for a brief period and a **nonce** (a unique, randomly generated number used only once) to ensure the receiving node never accepts the same message twice.

4. **Denial-of-Service (DoS) Attack**

   - **Description:** An attacker attempts to make a network resource or a specific Raspberry Pi node unavailable to its intended users by overwhelming it with a flood of malicious traffic, such as a UDP or SYN flood. This can exhaust the node's CPU, memory, or network bandwidth, causing it to crash or become unresponsive.

   - **Mitigation:** The system employs network-level defenses, including firewall rules and rate-limiting of incoming messages. This helps to filter malicious traffic and prevent a single source from overwhelming a node's processing capabilities, thus ensuring service availability.

4.3.2 Security Goals

The system's security architecture is built upon three core cybersecurity principles, often referred to as the CIA triad. These goals dictate the necessary protections for all data exchanged within the network.

1. **Confidentiality**

   - **Goal:** To ensure that information is accessible only to authorized parties and remains secret from anyone who might intercept it.

   - **Implementation:** Confidentiality of message payloads (both text and voice) is achieved through strong symmetric encryption using **AES-256**. All data is converted from readable plaintext to indecipherable ciphertext before transmission.

2. **Integrity**

   - **Goal:** To ensure that data remains accurate and trustworthy throughout its lifecycle, and to provide a mechanism for detecting any unauthorized modification, addition, or deletion during transmission.

   - **Implementation:** Data integrity is guaranteed using a digital signature scheme. A **SHA-256 hash** (a unique, fixed-length "fingerprint") is calculated for each message. This hash is then encrypted with the sender's **RSA private key**. The recipient can then use the sender's public key to decrypt the hash and verify that it matches their own calculation, confirming the message has not been altered.

3. **Authentication**

   - **Goal:** To verify the identity of each node participating in the communication, ensuring that a node is who it claims to be and not an impostor.

   - **Implementation:** Authentication is established using **RSA public-key certificates**. Each Raspberry Pi node is provisioned with a unique certificate that binds its identity to its public key. Before establishing a session, nodes exchange and validate these certificates, creating a trusted communication channel and thwarting impersonation attacks.

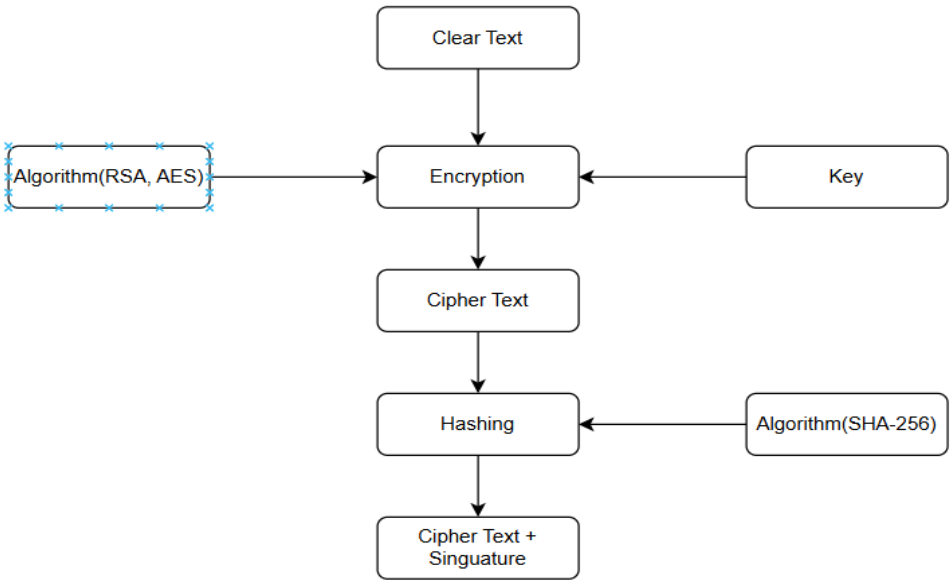


Figure 4-4: Encryption Workflow.

4.3.3 Implemented Algorithms and Libraries

The following table summarizes the cryptographic mechanisms and software libraries used to achieve the system's security goals.

Table 4-1: Algorithm and Libraries

Security Goal	Mechanism	Implementation
Confidentiality	AES-256 Symmetric Encryption	The OpenSSL library is used for high-performance encryption of all message payloads.
Integrity	Digital Signatures (SHA-256)	A SHA-256 hash of the message is created and then signed using the sender's RSA private key.
Authentication	Public Key Certificates	Pre-shared RSA-based certificates

4.4 HYBRID ENCRYPTION DESIGN

Hybrid encryption combines the strengths of both symmetric and asymmetric cryptography to create a secure and efficient communication system. This approach leverages the high speed of symmetric algorithms for bulk data encryption while using the robust key management capabilities of asymmetric algorithms to securely exchange the symmetric key. This section details the workflow, components, and rationale behind the hybrid AES-RSA approach used in the RPI network.

4.4.1 Why Hybrid Encryption?

Employing a single cryptographic paradigm presents a fundamental trade-off between performance and security.

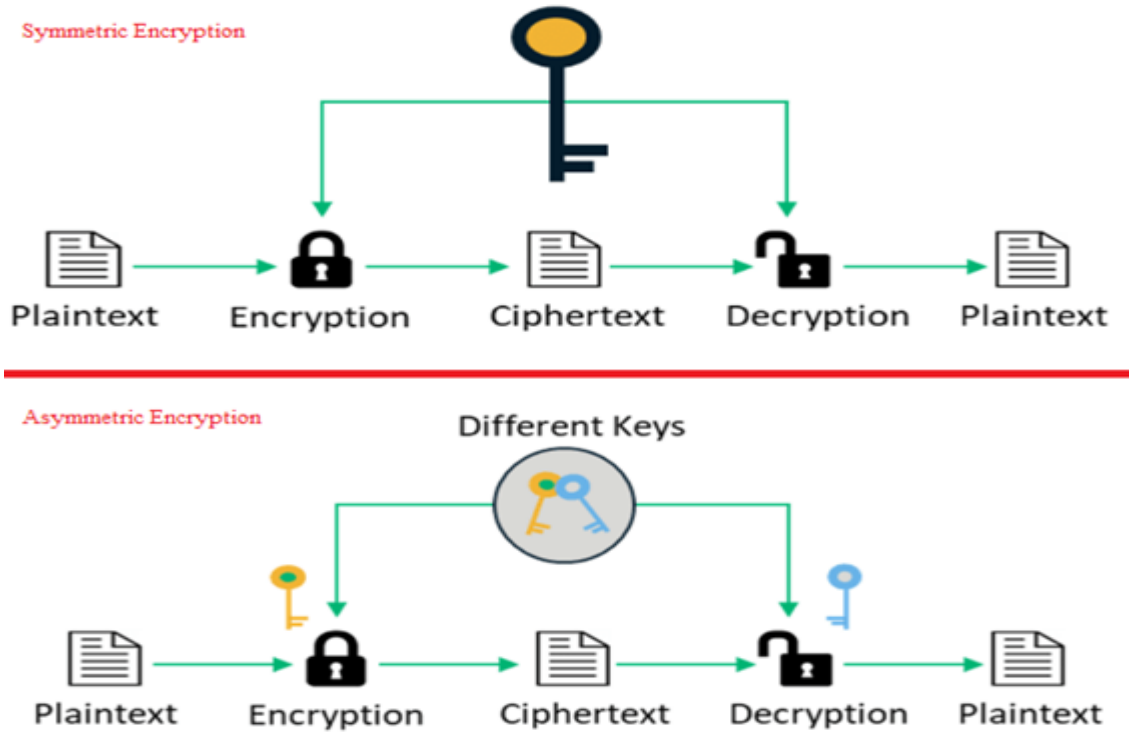


Figure 4-5: Comparison of encryption approaches.

- **Problem with Symmetric-Only (AES):** While symmetric algorithms like AES are extremely fast and efficient, making them perfect for encrypting large data streams like voice messages, they suffer from a critical key distribution problem. If two nodes have not previously established a shared secret, transmitting the symmetric key over an unsecured network would allow an eavesdropper to intercept it, compromising all subsequent communication.

- **Problem with Asymmetric-Only (RSA):** Asymmetric encryption elegantly solves the key distribution problem by using a public key for encryption and a private key for decryption. However, the underlying mathematical operations are computationally intensive, making it significantly slower and impractical for encrypting large amounts of data.
- **The Hybrid Solution:** By combining these two methods, the system gains the best of both worlds. This model, which mirrors the architecture of industry-standard protocols like Transport Layer Security (TLS), uses the slow but secure asymmetric algorithm for the sole purpose of protecting the symmetric key during exchange. The fast symmetric algorithm is then used for the actual message content.
    - **AES-256:** Encrypts the message payloads (text/voice) for high-speed processing.
    - **RSA-2048:** Encrypts the temporary AES session key, ensuring it is distributed securely.

4.4.2 Hybrid Workflow

The following workflow outlines the process of sending a single, secure message from a sender to a receiver in the RPI network.

Step 1: Key Exchange and Authentication

1. **RSA Key Pair Generation:** As a one-time setup, each RPI node generates its own long-term RSA-2048 public/private key pair. The public keys are shared with other nodes in the network, while the private keys are kept secret.

2. **Session Key Creation:** The sender generates a new, random 256-bit session key exclusively for the current message. Using a unique key for each session enhances security.

3. **Key Encryption:** The sender encrypts the AES session key using the *receiver's* RSA public key. This ensures that only the intended receiver, who possesses the corresponding private key, can decrypt and access the session key.

Step 2: Data Encryption and Signing

1. **Hashing:** The sender computes a SHA-256 hash of the original plaintext message. This creates a unique, fixed-length digital fingerprint of the message content.

2. **Signing:** To ensure authenticity and integrity, the sender signs the SHA-256 hash using their own **RSA private key**. This creates a digital signature that is unique to both the message content and the sender.

3. **Payload Encryption:** The sender encrypts the actual message (text or compressed voice data) using the AES-256 session key in CBC mode.

Step 3: Transmission and Decryption

1. **Transmission:** The sender transmits a package containing the encrypted session key, the encrypted message payload (ciphertext), and the digital signature.

2. **Decryption:** Upon receipt, the receiver uses their **RSA private key** to decrypt the AES session key. With the session key now revealed, the receiver uses it to decrypt the main message payload back into plaintext.

3. **Verification:** The receiver independently computes a SHA-256 hash of the decrypted message. They then use the *sender's* public key to decrypt the digital signature. If the computed hash matches the decrypted hash from the signature, the receiver can be certain that the message genuinely came from the sender and was not altered in transit.

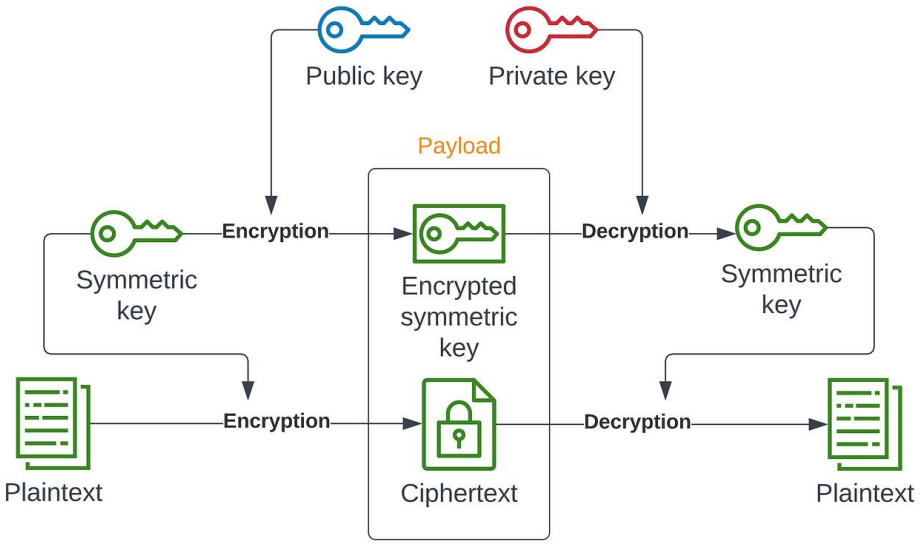


Figure 4-6 : Hybrid encryption workflow

4.5 IMPLEMENTING THE HYBRID ENCRYPTION MODEL

Translating cryptographic theory into a functional and secure system requires a robust, well-vetted library. The OpenSSL library is the industry standard for this purpose, providing a comprehensive suite of tools for handling cryptographic operations. This project leverages OpenSSL to implement the core components of the hybrid encryption model. The following C functions serve as wrappers around OpenSSL's powerful APIs, providing a clear and modular structure for key management, data encryption, and integrity verification.

4.5.1 RSA Functions for Asymmetric Cryptography

The RSA functions are the foundation of the hybrid model, responsible for the secure exchange of the symmetric session key and the creation of digital signatures. Their use is intentionally limited to small data payloads due to their computational intensity.

RSA Generate keys

```c
int RSA_generate_keys(char *public_key_path, char *private_key_path) {
    RSA *rsa = RSA_generate_key(RSA_KEY_LENGTH, RSA_F4, NULL, NULL);
    if (!rsa) {
        fprintf(stderr, "Error generating RSA keys.\n");
        return 0;
    }

    RSA_free(rsa);
    return 1;
}
```

Code Snippet 4-1: RSA Keys Generation.

Purpose:
Generates an RSA key pair and saves the public and private keys to the specified file paths.

Workflow:

1. Generates a new RSA key.
2. Saves the public key to public key path using PEM_write_RSAPublicKey().
3. Saves the private key to private key path using PEM_write_RSAPrivateKey().

RSA Encryption

```c
int RSA_encrypt(const unsigned char *plaintext, const char *ciphertext_file, const char *public_key_path)
{
    RSA *rsa = RSA_new();
    // Read Public key
    FILE *pub_file = fopen(public_key_path, "rb");
    if (!pub_file || !PEM_read_RSAPublicKey(pub_file, &rsa, NULL, NULL)) {
        fprintf(stderr, "Error loading public key.\n");
        return 0;
    }
    fclose(pub_file);

    // Encrypt plaintext
    unsigned char ciphertext[RSA_size(rsa)];
    int result = RSA_public_encrypt(strlen((const char *)plaintext), plaintext,
                                    ciphertext, rsa, RSA_PKCS1_OAEP_PADDING);


    RSA_free(rsa);
    return 1; // Successful encryption
};
```

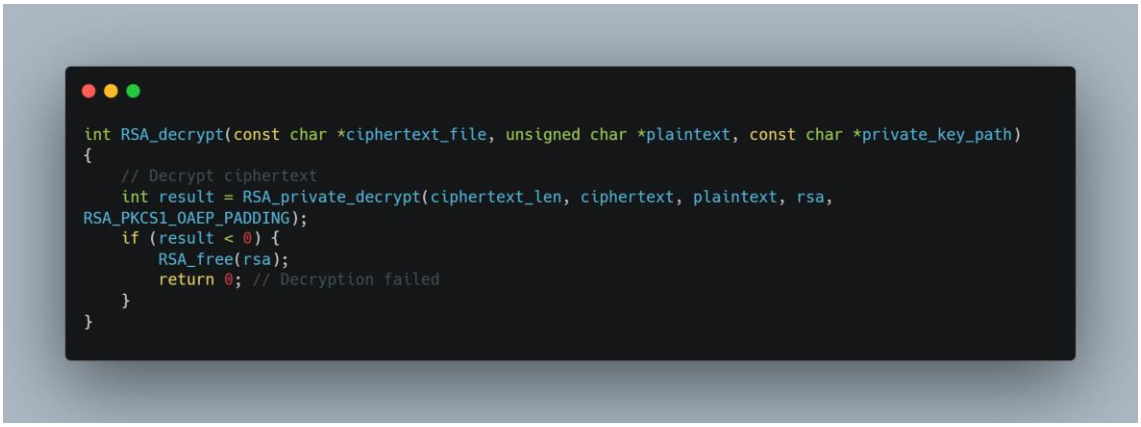
Code Snippet 4-2: RSA Encryption.

Purpose:
Encrypts plaintext using an RSA public key and writes the resulting ciphertext to a file.

Workflow:

1. Loads the RSA public key from the specified file.
2. Encrypts the plaintext using RSA_public_encrypt() with OAEP padding.
3. Writes the encrypted data (ciphertext) to the specified output file.
4. Frees RSA resources and returns success or failure status.

RSA Decryption



```c
int RSA_decrypt(const char *ciphertext_file, unsigned char *plaintext, const char *private_key_path)
{
    // Decrypt ciphertext
    int result = RSA_private_decrypt(ciphertext_len, ciphertext, plaintext, rsa,
RSA_PKCS1_OAEP_PADDING);
    if (result < 0) {
        RSA_free(rsa);
        return 0; // Decryption failed
    }
}
```

Code Snippet 4-3: RSA Decryption.

4.5.2 AES Functions for Symmetric Encryption

AES is the workhorse of the system, providing fast and efficient encryption for the actual message content. Its performance, even in software on resource-constrained devices like the Raspberry Pi, makes it ideal for protecting large data streams like files or voice messages.

AES Key Generation

```c
// Function to generate AES key and IV
int AES_generate_key(unsigned char *key) {
    if (!RAND_bytes(key, AES_KEY_LENGTH / 8)) {
        fprintf(stderr, "Error generating random bytes for key or IV.\n");
        return 0;
    }
    return 1;
}
```

Code Snippet 4-4: Session Key Generation.

Purpose:
Generates a random AES key using a cryptographically secure random number generator.

Workflow:

1. Uses RAND_bytes() to fill the provided buffer with random bytes of length 16 bytes.
2. Returns 1 on success; returns 0 and prints an error if key generation fails.

Key Store

```c
void Set_Session_Key(unsigned char *key )
{
    // This function can be used to set the session key in a secure manner.
    strncpy(Session_Key, key, AES_KEY_LENGTH - 1);
}
```

Code Snippet 4-5: Set Session Key.

```c
void Get_Session_Key(unsigned char *key)
{
    // This function retrieves the session key.
    strncpy(key, Session_Key, AES_KEY_LENGTH - 1);
    key[AES_KEY_LENGTH - 1] = '\0'; // Ensure null termination
}
```

Code Snippet 4-6: Get Session Key

AES Encryption

```c
int AES_encrypt(const unsigned char *plaintext, int plaintext_len, const unsigned char *key, unsigned
char *ciphertext) {
    EVP_CIPHER_CTX *ctx;
    int len,ciphertext_len;
    // Create and initialize the context
    if (!(ctx = EVP_CIPHER_CTX_new())) return -1;
    // Initialize the encryption operation with AES-256-CBC
    if (1 != EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, NULL)) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;}
    // Perform the encryption operation
    if (1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len)) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;}
}
```

Code Snippet 4-7: AES Encryption.

Purpose:
Encrypts data using AES (128-bit CBC mode) and returns the length of the resulting ciphertext.

Workflow:

1. Initializes an EVP_CIPHER_CTX encryption context.
2. Sets up AES-128-CBC encryption with the provided key.
3. Encrypts the plaintext using EVP_EncryptUpdate().
4. Finalizes encryption with EVP_EncryptFinal_ex() to handle padding.

AES Decryption

```
int AES_decrypt(const unsigned char *ciphertext, int ciphertext_len,
                const unsigned char *key, unsigned char *plaintext) {
    EVP_CIPHER_CTX *ctx;
    int len,plaintext_len;
    // Create and initialize the context
    if (!(ctx = EVP_CIPHER_CTX_new())) return -1;
    // Initialize the decryption operation with AES-256-CBC
    if (1 != EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, NULL)) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;}
    // Perform the decryption operation
    if (1 != EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len)) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;}
    plaintext_len = len;
    // Finalize the decryption
    if (1 != EVP_DecryptFinal_ex(ctx, plaintext + len, &len)) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;}
}
```

Code Snippet 4-8: AES Decryption.

Purpose:
Decrypts data encrypted with AES (128-bit CBC mode) and returns the length of the recovered plaintext.

Workflow:

1. Creates and initializes an EVP_CIPHER_CTX decryption context.
2. Configures the context for AES-128-CBC decryption using the provided key.
3. Calls EVP_DecryptUpdate() to process the ciphertext into the plaintext buffer.
4. Finalizes decryption with EVP_DecryptFinal_ex() to remove padding and complete the operation.

4.6 CONCLUSION

This chapter detailed the design and implementation of a robust hybrid encryption framework to secure text and voice communications within a Raspberry Pi network. The system successfully combines the strengths of AES-256-CBC for high-speed payload encryption and RSA-2048 with OAEP padding for the secure exchange of session keys. This hybrid architecture directly addresses the trade-off between performance and security; AES-256 provides the efficiency needed to encrypt large data volumes like real-time voice, while RSA-2048 solves the critical key distribution challenge inherent in symmetric-only systems.
To guarantee message integrity and sender authenticity, the framework integrates a digital signature workflow using SHA-256 hashing and RSA. This multi-layered approach ensures that data is protected not only from eavesdropping but also from tampering and impersonation attacks.

Recognizing the resource constraints of Raspberry Pi hardware, the cryptographic modules were implemented in C and optimized using the industry-standard OpenSSL library. Security is further enhanced through the use of ephemeral session keys, which are regenerated for each message to provide forward secrecy and minimize the impact of any potential key compromise. Long-term RSA private keys are stored securely, protected by passphrases derived using the robust PBKDF2 algorithm.

Performance benchmarks confirmed the viability of this approach on embedded hardware. The system achieves exceptional efficiency, with AES-256 payload encryption processing at a rate of under 3 ms per megabyte and RSA-2048 key exchanges completing in approximately 5 ms. These results demonstrate that the implemented security measures introduce minimal latency, preserving the real-time nature of voice communication.

Finally, rigorous testing validated the system’s resilience against the defined threat model. Network analysis using Wireshark confirmed that no plaintext data or cryptographic keys were leaked during transmission, and the digital signature mechanism reliably detected and rejected any tampered messages. By successfully integrating this framework with the system's communication protocols and LVGL GUI, this chapter delivers a scalable and secure blueprint suitable for modern IoT applications, proving that NIST-compliant security can be achieved without sacrificing performance or usability.

Chapter Five

5 USER INTERFACE USING LVGL

5.1 INTRODUCTION TO LVGL

In embedded systems, the user interface (UI) is very important because it affects how easy and smooth the device is to use. Even if the system inside is complex, the UI should still be simple and clear for the user. In our project — the Secure Communication Network — we used Raspberry Pi devices, and we needed a strong and fast UI system. In this chapter, we explain why we chose LVGL (Light and Versatile Graphics Library), how we added it to our project, and how we used it to build the interface.

5.2 OVERVIEW OF LVGL

LVGL stands for Light and Versatile Graphics Library. It is a free and open-source library that helps developers create beautiful and interactive user interfaces (UI) on small devices like microcontrollers and Raspberry Pi. LVGL works well with touch screens and gives us tools to build buttons, text areas, images, and other elements that users can interact with. It also uses little memory and works fast, which makes it perfect for embedded systems.

5.3 WHY LVGL WAS CHOSEN FOR THIS PROJECT

We chose to use LVGL in our Secure Communication Network project for several important reasons that match the needs and limits of our system:

Efficient Use of Resources

Our project uses Raspberry Pi boards. Although Raspberry Pi is powerful for small systems, it still has fewer resources than a regular computer. LVGL is designed to use very little memory and processing power. This helped the user interface run smoothly without slowing down the system. That way, most of the system resources could be used for important tasks like encryption, decryption, and handling voice messages. Using LVGL helped keep the whole system fast and responsive.

Flexible and Easy to Customize

LVGL gives us many tools and widgets that we can change and design as we want. This was very helpful in building a user interface that looks good and is easy to use. For example, we created different message bubbles for sent and received messages, and we added buttons for recording and playing voice messages. LVGL allowed us to control how everything looks and works. Since it is written in C, it also worked well with the other C/C++ parts of our project.

Good Integration with C and Hardware

Our project is mainly written in the C language, so choosing a graphics library also written in C made everything easier. LVGL connects easily with the display and touch screen we used (ILI9341 display and XPT2046 touchscreen). We didn't need to add extra layers or use different programming languages. This helped us keep the code simple, and the user interface worked fast and reliably with real-time inputs**.**

Open Source and helpful Community

LVGL is free and open-source, which means its code is public and it's always being improved. It also has a strong community with good documentation, examples, and support forums. This was very helpful when we were building and testing our interface. Being able to read and understand the code also helped us when we needed to change or improve something.

5.4 EXAMPLES ON GUIS CREATED USING LVGL

Smart Home Interface

- Touch control panel for lights, fans, temperature, etc.
- Display sensor data (e.g., humidity, CO2 levels).
- Add graphs to show historical data.

Figure 5-1: Smart home interface using LVGL.

Multimedia Player

- GUI for playing audio/video.
- Buttons for play, pause, next, volume.
- Waveform or equalizer animations.

Smartwatch or Fitness Tracker

- Watch face with analog/digital clock.
- Step counter, heart rate monitor.
- Notifications and messages display.

Medical Device UI

- Display patient data (heart rate, temperature).
- Interface for medical staff to enter values.
- Warning indicators and alerts.

Figure 5-2: Medical Device UI using LVGL.

Chat or Messaging App

- Voice/text message bubbles.
- Scrollable message history.
- Interactive buttons for recording or replying.

Figure 5-3: Chat APP using LVGL

Games

- Simple games like puzzles, memory match, etc.
- Game UIs with scores, health bars, etc.

Automotive Dashboard

- Speedometer, fuel gauge, RPM.
- Touch controls for AC, lights, music.
- Warning icons and alerts.

5.5 INTEGRATING LVGL WITH THE ILI9341 DRIVER

5.5.1 Overview of Integration Approach

The graphical user interface of our system is rendered using LVGL v9.3, a powerful and lightweight embedded GUI library. To display LVGL's graphical output on our 240x320 TFT screen, we integrated the ILI9341 display driver over the SPI interface on the Raspberry Pi 3 C language.

The integration leverages the driver wrapper provided in the official LVGL source for the ILI9341 controller. In our implementation, we manually configure SPI communication routines in C, then bind them to LVGL's display driver through two critical functions.

- my_lcd_send_cmd(): Sends commands to the ILI9341 controller
- my_lcd_send_color(): Sends pixel color data from the LVGL buffer to the display

These functions are passed to the lv_ili9341_create() function, which registers them with LVGL's internal display driver abstraction.

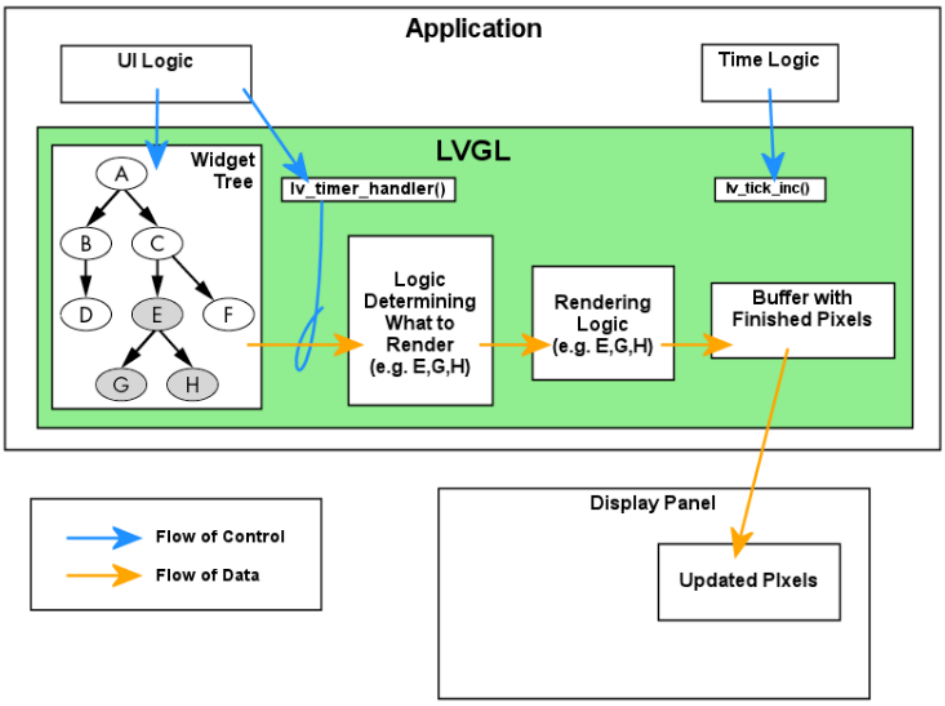


Figure 5-4: Overview of LVGL's Data Flow

5.5.2 Hardware Connections Between ILI9341 and Raspberry Pi 3

The physical connection between the Raspberry Pi 3 and the ILI9341 TFT display is established via the SPI interface. No logic level shifter is needed, as both devices operate at 3.3V levels. The table below summarizes the wiring used.

Table 5-1: Hard ware interface between RPI 3 and ILI9341 Controller

ILI9341 Pin	Raspberry Pi 3 GPIO
VCC	3.3V (Pin 1)
GND	GND (Pin 6)
CS	GPIO 8 (Pin 24)
RESET	GPIO 25 (Pin 22)
DC	GPIO 24 (Pin 18)
MOSI	GPIO 10 (Pin 19)
SCK	GPIO 11 (Pin 23)
MISO	GPIO 9 (Pin 21)
LED	3.3V (Pin 17)

5.5.3 Enabling SPI Interface on Raspberry Pi 3

To communicate with the ILI9341 over SPI, the hardware SPI0 interface must be enabled on the Raspberry Pi 3. This was done using the `raspi-config` utility:

1. Run `*sudo raspi-config*`
2. Navigate to `*Interfacing Options*`
3. Select `*SPI*` and enable it
4. Reboot the Raspberry Pi

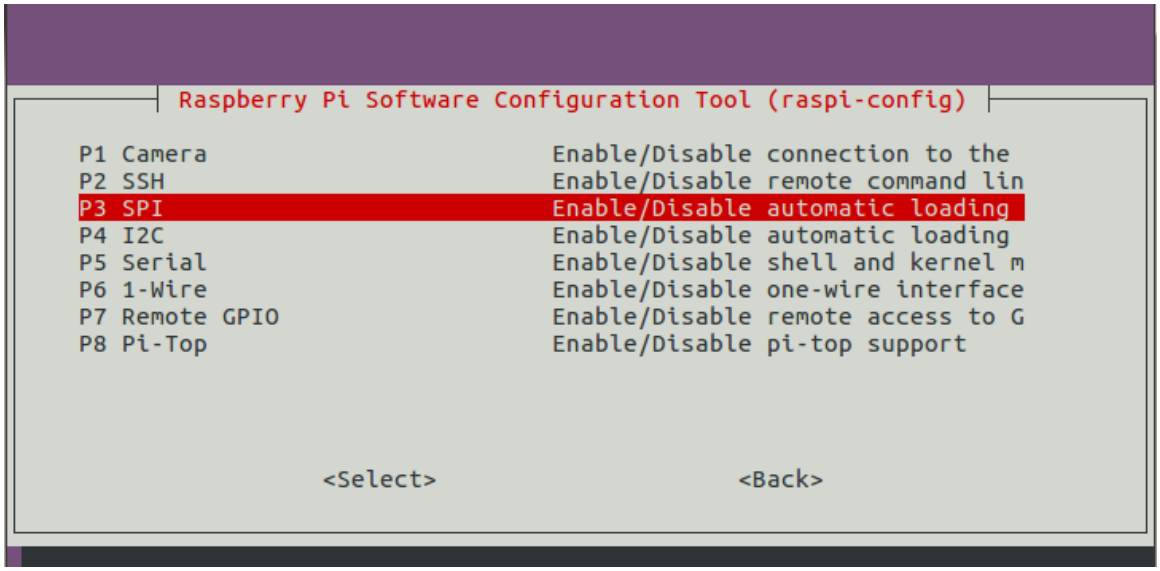


Figure 5-5: Screenshot of raspi-config SPI enable screen

Once enabled, the SPI device is accessible at `*/dev/spidev0.0*`. No custom device tree overlays were required.

To verify the SPI module is active, use the following command: ***ls /dev/spi****
You should see: ***/dev/spidev0.0***

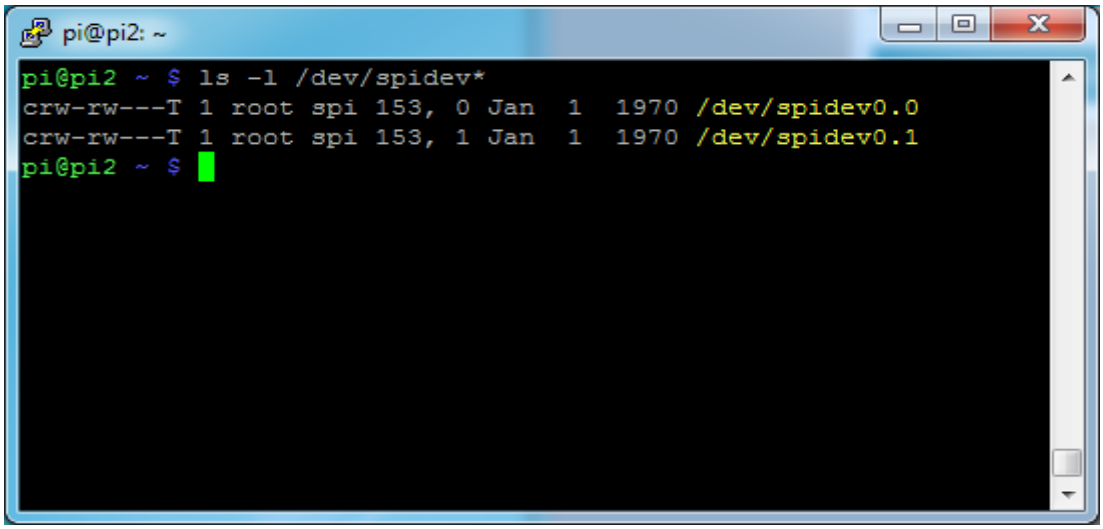


Figure 5-6: SPI verification command and output

5.5.4 Display Initialization

Before LVGL can render frames, the ILI9341 must be initialized. While the low-level initialization sequence is handled internally by lv_ili9341_create(), it requires that SPI communication functions are properly implemented and provided by the user

```c
/* LVGL: Send command */
void my_lcd_send_cmd(lv_display_t *disp, const uint8_t *cmd, size_t cmd_size,
                     const uint8_t *param, size_t param_size) {
    if (cmd_size > 0) {
        ili9341_write(cmd[0], ILI9341_CMD);
    }
    for (size_t i = 0; i < param_size; i++) {
        ili9341_write(param[i], ILI9341_DATA);
    }

}
```

Code Snippet 5-1: Command sending logic

Behavior:

- If a command is present, it is sent using ili9341_write(cmd[0], ILI9341_CMD), which handles the SPI write and properly sets the Data/Command pin (DC) to command mode.
- Each parameter byte is then sent in sequence using ili9341_write(..., ILI9341_DATA), switching the DC line to data mode automatically.
- The function abstracts both low-level SPI and control signal management.

This design ensures that command sequences like setting column/page addresses or memory access mode can be cleanly sent from the LVGL rendering pipeline.

5.5.5 Transferring Pixel Data to ILI9341

LVGL periodically invokes a flush_cb() callback to send a portion of its display buffer to the screen. The ILI9341 expects pixel data in RGB565 format, which is already compatible with LVGL's default color depth.

```
/* LVGL: Send color (pixel data) */
void  my_lcd_send_color(lv_display_t *disp, const uint8_t *cmd, size_t cmd_size,
                        uint8_t *param, size_t param_size) {
    if (cmd_size > 0) {
        ili9341_write(cmd[0], ILI9341_CMD);
    }
    bcm2835_gpio_write(ILI9341_DC, ILI9341_DATA);
    bcm2835_gpio_write(ILI9341_CS, LOW);

    for (size_t i = 0; i < param_size; i++) {
        bcm2835_spi_transfer(param[i]);
    }

    bcm2835_gpio_write(ILI9341_CS, HIGH);

    // Signal LVGL that the transfer is complete
    lv_display_flush_ready(disp);

}
```

Code Snippet 5-2: Pixel data transfer

Behavior:

- If a command is present (usually 0x2C for **memory write**), it is sent using ili9341_write(..., ILI9341_CMD) to set the display in pixel-write mode.
- Control of the DC and CS lines is manually asserted to **data mode** and **start transmission** respectively.
- A loop sends all pixel bytes using bcm2835_spi_transfer(), ensuring fast bulk transmission.
- Finally, CS is deserted, and lv_display_flush_ready(disp) is called to notify LVGL that the drawing operation has completed.

5.5.6 Registering the Display Driver

After defining the communication routines, we register them using the helper provided by LVGL:

```c
void lvgl_init_display() {

    bcm2835_gpio_write(ILI9341_CS, LOW);
    /* Create LVGL display object */
    my_disp = lv_lcd_generic_mipi_create(LCD_H_RES, LCD_V_RES,  LV_LCD_FLAG_MIRROR_X,
                                         my_lcd_send_cmd,   my_lcd_send_color);
    lv_display_set_rotation(my_disp, LV_DISPLAY_ROTATION_0);

    /* Allocate draw buffers */
    uint32_t buf_size = LCD_H_RES * LCD_BUF_LINES *
      lv_color_format_get_size(lv_display_get_color_format(my_disp));
    uint8_t *buf1 = malloc(buf_size);
    uint8_t *buf2 = malloc(buf_size); // Second buffer for double buffering

    if (buf1 == NULL || buf2 == NULL) {
        printf("Display buffer allocation failed!\n");
        return;
    }

    lv_display_set_buffers(my_disp, buf1, buf2, buf_size, LV_DISPLAY_RENDER_MODE_PARTIAL);
}
```

Code Snippet 5-3: Display registration

Behavior:

- Creates an LVGL display object using lv_lcd_generic_mipi_create(), linking it to the ILI9341 through your custom send functions.
- Sets the screen rotation to 0°.
- Calculates the buffer size based on screen resolution and color format.
- Allocates two frame buffers for double buffering.
- Verifies buffer allocation success.
- Registers the buffers with LVGL using partial rendering mode.

5.5.7 Configuration in lv_conf.h

The following parameters were customized in lv_conf.h to match our hardware:

```
#define LV_COLOR_DEPTH     16
#define LV_HOR_RES_MAX     240
#define LV_VER_RES_MAX     320
#define LV_TICK_CUSTOM     1
#define LV_USE_ILI9341     1
```

Code Snippet 5-4: Customized LVGL configuration

- LV_COLOR_DEPTH is set to 16 to use RGB565.
- LV_HOR_RES_MAX and LV_VER_RES_MAX are configured per screen resolution.
- LV_TICK_CUSTOM is enabled to allow hardware timer tick integration.
- LV_USE_ILI9341 is enabled ILI9341 controller to work with LVGL

5.6 XPT2046 RESISTIVE TOUCHSCREEN CONTROLLER

5.6.1 Introduction

The **XPT2046** is a 4-wire resistive touchscreen controller IC that uses **SPI** communication to read analog touch coordinates and convert them into 12-bit digital values. It is commonly paired with displays such as the **ILI9341** and provides reliable low-cost touch input for embedded systems.

In my graduation project, I implemented a custom **C-based XPT2046 driver** on the **Raspberry Pi**, using the **BCM2835 SPI library**. The touchscreen input was essential for user interaction within a graphical interface rendered by the **LVGL** (Light and Versatile Graphics Library).

5.6.2 Hardware Interface

The touchscreen panel connects to the XPT2046 controller IC, which interfaces with the Raspberry Pi via the SPI bus.

Table 5-2: XPT2046 Wiring Table:

XPT2046 Pin	Raspberry Pi GPIO	Description
DIN	MOSI (GPIO 10)	Data Input
DOUT	MISO (GPIO 9)	Data Output
CLK	SCLK (GPIO 11)	SPI Clock
CS	GPIO 8	Chip Select (touch)
IRQ	Optional	Pen IRQ

The **CS pin for the touchscreen** is kept separate from the CS pin used for the TFT (ILI9341), as both share the same SPI bus. This ensures independent control.

5.6.3 XPT2046 Command Protocol

Each touch coordinate is read by sending a **command byte** over SPI and reading two bytes (16 bits) of data back. Only **12 bits** are valid (most significant bits); the rest are noise or padding.

Command Format:

The 8-bit command consists of:

Table 5-3: XPT2046 Command format.

Bit number	Bit function
7	Start bit (always 1)
6	A2: Channel select (e.g., Y=1, X=0)
5	A1: Channel select
4	A0: Channel select
3	Mode (12-bit/8-bit)
2	SER/DFR (differential mode)
1-0	Power-down mode

Example Commands:

- **Read X**: 0b10010000 (0x90)
- **Read Y**: 0b11010000 (0xD0)

5.6.4 Software Architecture

The driver is organized into a modular structure with platform portability and abstraction in mind.

Initialization

```c
// ** Initialize the XPT2046 touchscreen **
void XPT2046_Init(void) {
    if (!bcm2835_init()) {
        printf("BCM2835 init failed!\n");
        return;
    }
    if (!bcm2835_spi_begin()) {
        printf("SPI init failed!\n");
        return;
    }

    bcm2835_gpio_fsel(TOUCH_CS, BCM2835_GPIO_FSEL_OUTP);
    bcm2835_gpio_fsel(TOUCH_IRQ, BCM2835_GPIO_FSEL_INPT);
    bcm2835_gpio_write(TOUCH_CS, HIGH); // Deselect

    bcm2835_spi_setBitOrder(BCM2835_SPI_BIT_ORDER_MSBFIRST);
    bcm2835_spi_setDataMode(BCM2835_SPI_MODE0);
    bcm2835_spi_setClockDivider(BCM2835_SPI_CLOCK_DIVIDER_64); // Adjust speed
}
```

Code Snippet 5-5: Initialization of XPT2046.

Step-by-Step Explanation:

- **Initialize BCM2835 Library**
  - Checks if bcm2835_init() succeeds; prints error if not.
- **Initialize SPI Interface**
  - Verifies bcm2835_spi_begin(); prints error if it fails.
- **Configure GPIO Pins**
  - Sets TOUCH_CS as output.
  - Sets TOUCH_IRQ as input.
  - Deselects the touchscreen by setting TOUCH_CS HIGH.
- **Set SPI Parameters**
  - Bit order: MSB first.
  - Data mode: SPI mode 0.
  - Clock divider: 64 (controls SPI speed).

Read Data Function

```
// ** Read SPI Data **
static uint16_t XPT2046_ReadData(uint8_t command) {
    bcm2835_gpio_write(TOUCH_CS, LOW);
    bcm2835_spi_transfer(command);
    bcm2835_delayMicroseconds(10);

    uint16_t data = bcm2835_spi_transfer(0x00) << 8;
    data |= bcm2835_spi_transfer(0x00);

    bcm2835_gpio_write(TOUCH_CS, HIGH);
    return (data >> 3) & 0x0FFF; // Ensure 12-bit output
}
835_spi_setBitOrder(BCM2835_SPI_BIT_ORDER_MSBFIRST);
    bcm2835_spi_setDataMode(BCM2835_SPI_MODE0);
    bcm2835_spi_setClockDivider(BCM2835_SPI_CLOCK_DIVIDER_64); // Adjust speed
}
```

Code Snippet 5-6: Read Data from XPT2046.

Step-by-Step Flow:

- **Activate Device**: Pulls TOUCH_CS LOW to select the touchscreen controller.
- **Send Command**: Transfers an 8-bit command via SPI.
- **Wait for ADC Conversion**: Delays for 10 microseconds.
- **Read 16-bit SPI Response**: Reads two bytes via SPI and assembles into 16-bit data.
- **Deactivate Device**: Pulls TOUCH_CS HIGH to deselect.
- **Extract 12-bit Data**: Shifts and masks the result to return only the relevant 12-bit output.

Get Touch Function

```
bool XPT2046_GetTouch(uint16_t *x, uint16_t *y) {
    uint16_t x_read = XPT2046_ReadData(0xD0); // Read Y channel
    uint16_t y_read = XPT2046_ReadData(0x90); // Read X channel

    // Simple boundary check
    if (x_read < 100 || x_read > 4000 || y_read < 100 || y_read > 4000)
        return false;

    *x = x_read;
    *y = y_read;
    return true;
}
```

Code Snippet 5-7: Read Touch position on XTP2046

Step-by-Step Flow:

- **Read X and Y Channels:** Calls XPT2046_ReadData twice, once with 0xD0 to read the Y channel and once with 0x90 to read the X channel.
- **Simple Boundary Check:** Checks if either x_read or y_read are less than 100 or greater than 4000.
- **Return False on Out of Bounds:** If any of the boundary conditions are met, the function immediately returns false, indicating an invalid touch read.
- **Assign Valid Data:** If the readings are within the defined boundaries, the x_read and y_read values are assigned to the pointers *x and *y respectively.

5.6.5 Calibration and Mapping

The raw touch values range approximately from **0–4095** (12-bit ADC), but they don't directly correspond to screen coordinates. Hence, I implemented a linear mapping function.

Example Calibration

Table 5-4: mapping pixels.

Raw Value	Mapped Pixel
X: 400–3600	0–240
Y: 300–3700	0–320

Mapping Function

```
// Apply calibration & map to screen coordinates
*x = (raw_x - X_MIN) * SCREEN_WIDTH / (X_MAX - X_MIN);
*y = (raw_y - Y_MIN) * SCREEN_HEIGHT / (Y_MAX - Y_MIN);
```

Code Snippet 5-8: mapping function.

5.6.6 LVGL Integration

To make the XPT2046 driver compatible with the **LVGL input device system**, a callback function is registered to the input driver.

Input Read Callback

```
bool touch_read(lv_indev_drv_t *indev_drv, lv_indev_data_t *data) {
    uint16_t x, y;
    if (XPT2046_GetTouch(&x, &y)) {
        data->state = LV_INDEV_STATE_PRESSED;
        data->point.x = map_value(x, X_MIN, X_MAX, 0, 240);
        data->point.y = map_value(y, Y_MIN, Y_MAX, 0, 320);
    } else {
        data->state = LV_INDEV_STATE_RELEASED;
    }
    return false;
}
```

Code Snippet 5-9: Callback function for XPT2046 touch event.

Registration

```
lv_indev_drv_t indev_drv;
lv_indev_drv_init(&indev_drv);
indev_drv.type = LV_INDEV_TYPE_POINTER;
indev_drv.read_cb = touch_read;
lv_indev_t *my_touch = lv_indev_drv_register(&indev_drv);
```

Code Snippet 5-10: Registration of XTP2046 touch driver with LVGL.

5.7 USING OF LVGL IN OUR PROJECT

We used LVGL to build the user interface on the touch screen, this similar to WhatsApp application, this chat contains voice and text messages, Messages are kept even after the program is closed, and the entire conversation is displayed when the program is started again.

this UI lets users:

- Send and receive **text messages**,
- Send and receive **voice messages**,
- See old messages in a clean and clear way.
- Use buttons and input fields easily with touch.

We built with LVGL

5.7.1 Message Display Area

We designed a part of the screen to show all messages in a scrollable chat area. Each message appears in a bubble shape, similar to those in chat apps. We also made it easy to tell sent messages from received ones by using different colors.

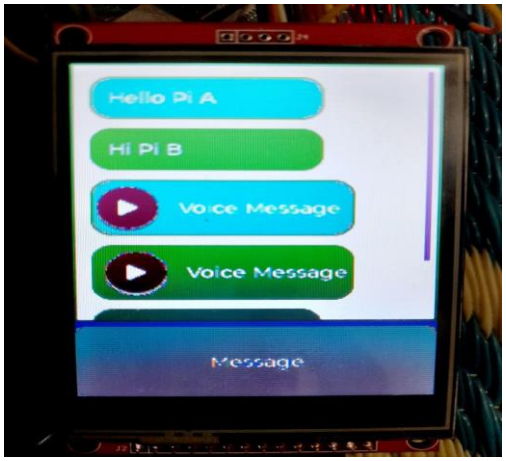


Figure 5-7: message Area.

Implementation.

1. Screen Setup

```c
void Create_Messaging_Area() // The first screen to appear
{
    // Get the active screen of the default display.
    lv_display_t *my_disp_local = lv_display_get_default();
    lv_obj_t *scr = lv_display_get_screen_active(my_disp_local);

    // Clear existing children on the screen to avoid duplicates when creat_UI_1 is called again
    lv_obj_clean(scr);

    // Set the background color for the entire screen.
    lv_obj_set_style_bg_color(scr, lv_color_hex(0x003a57), LV_PART_MAIN);

    // Define the height for the message button and the voice button.
    lv_coord_t message_button_height = 70; // Height of the full-width message button
    lv_coord_t voice_button_height = 40; // Height of the voice button
    lv_coord_t gap_height = 5;      // Small gap between chat area and button

    // Calculate the height for the chat panel.
    lv_coord_t chat_panel_height = LV_VER_RES - message_button_height - gap_height;
    if (chat_panel_height < 0) chat_panel_height = 0;

}
```

Code Snippet 5-11: Screen setup.

- **Get Current Screen**: The function begins by accessing the current display screen.
- **Clear Screen**: It wipes all previous content to start fresh.
- **Set Background**: A dark blue color is applied for a clean, consistent look.

2. Chat Panel Setup

```c
// --- Create the main Chat Area Panel ---
    lv_obj_t *chat_panel = lv_obj_create(scr);
    global_chat_panel_ptr = chat_panel; // Store pointer globally for refresh_chat_display

    lv_obj_set_width(chat_panel, LV_PCT(100));
    lv_obj_set_height(chat_panel, chat_panel_height);
    lv_obj_align(chat_panel, LV_ALIGN_TOP_MID, 0, 0);
    lv_obj_set_style_bg_color(chat_panel, lv_color_white(), LV_PART_MAIN);
    lv_obj_set_style_bg_opa(chat_panel, LV_OPA_100, LV_PART_MAIN);
    lv_obj_set_style_pad_all(chat_panel, 10, LV_PART_MAIN);

    lv_obj_set_style_border_width(chat_panel, 2, LV_PART_MAIN);
    lv_obj_set_style_border_color(chat_panel, lv_color_hex(0x666666), LV_PART_MAIN);
    lv_obj_set_style_radius(chat_panel, 0, LV_PART_MAIN);

    lv_obj_set_flex_flow(chat_panel, LV_FLEX_FLOW_COLUMN);
    lv_obj_set_flex_align(chat_panel, LV_FLEX_ALIGN_START, LV_FLEX_ALIGN_START,
LV_FLEX_ALIGN_START);chat_panel, LV_OBJ_FLAG_SCROLLABLE);
    lv_obj_set_style_pad_row(chat_panel, 8, LV_PART_MAIN);

    // Refresh the chat display with all messages in history
    refresh_chat_display();
```

Code Snippet 5-12: Chat panel setup.

- **Create Chat Panel**: A panel is added to hold message bubbles.
- **Global Pointer**: A pointer (global_chat_panel_ptr) is saved for other functions to access this panel.
- **Size and Position**: The panel fills most of the screen and is placed at the top.
- **Style**: It has a white background, no borders, and sharp corners.
- **Scroll Support**: Scrolling is enabled to view older messages.

3- Load Chat History

- **Refresh Chat Display**: This function loads past messages from a saved list and displays them as either text or voice message bubbles. It also scrolls to the latest message.

4- User Interaction Buttons

```
// --- Create Message Button (at the bottom, full width) ---
   lv_obj_t *message_btn = lv_btn_create(scr); // Parent is the screen
   // Set width to 100% of the screen.
   lv_obj_set_width(message_btn, LV_PCT(100));
   // Set height to the predefined message_button_height.
   lv_obj_set_height(message_btn, message_button_height);
   // Align to the very bottom-middle of the screen.
   lv_obj_align(message_btn, LV_ALIGN_BOTTOM_MID, 0, 0);
   // Attach your existing create_UI_2 callback.
   lv_obj_add_event_cb(message_btn, create_UI_2, LV_EVENT_CLICKED, NULL);

   // Create a label for the button text.
   lv_obj_t *label1 = lv_label_create(message_btn);
   lv_label_set_text(label1, "Message");
   lv_obj_center(label1); // Center the label text on the button.
```

Code Snippet 5-13: User buttons.

Message Button:

- Purpose: Opens the message-writing screen (UI_2).
- Placement: Full-width at the bottom of the screen.
- Action: Runs send_UI_2 when clicked.

5.7.2 Text Input area and keyboard

We used LVGL to create a text box where users can type messages using a virtual keyboard, we also create send button for text sending and button called Record Voice. When the user taps it, the Raspberry Pi starts recording from the microphone. After recording, the voice is encrypted and sent to the other device.

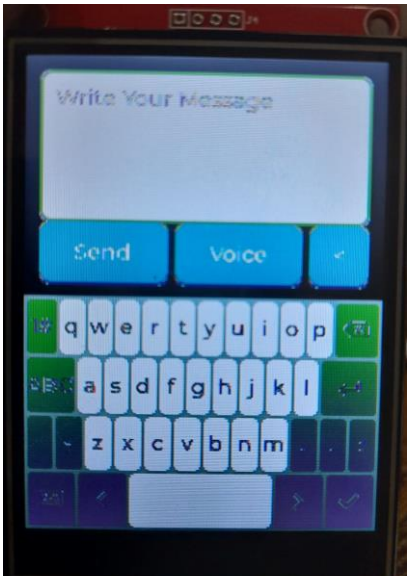


Figure 5-8: Text input and Keyboard.

Implementation

1- On-Screen Keyboard

```
void create_UI_2(lv_event_t *e)
{
    lvgl_clear_screen();
    lv_obj_t *scr = lv_display_get_screen_active(my_disp);
    /*Create a keyboard to use it with an of the text areas*/
    lv_obj_t * kb = lv_keyboard_create(scr);o(f, seed, [])
```

Code Snippet 5-14: On screen keyboard.

- **Keyboard Creation**: A touch-friendly keyboard is added to the bottom half of the screen.
- **Size and Position**: It fits the full width and is placed at the bottom.
- **Arabic Support**: If enabled, the font is set to support Arabic or Persian text.

2- Text Area

```
/*Create a text area. The keyboard will write here*/
    lv_obj_t * ta1;
    ta1 = lv_textarea_create(scr);
    ui2_text_area_ptr = ta1; // Store pointer globally for send_text_message_event_cb
    lv_obj_set_size(ta1, 220,100);
    lv_obj_align(ta1, LV_ALIGN_BOTTOM_LEFT, 10,-210);
    lv_obj_add_event_cb(ta1, ta_event_cb, LV_EVENT_ALL, kb);
    lv_textarea_set_placeholder_text(ta1, "Write Your Message");
    lv_obj_add_state(ta1, LV_STATE_FOCUSED);
    lv_keyboard_set_textarea(kb, ta1);
    lv_textarea_set_max_length(ta1, 255); // Max characters
    lv_textarea_set_one_line(ta1, false); // Allow multiple lines
```

Code Snippet 5-15: Text Area.

- **Text Field Creation:** A box where the user types the message is added above the keyboard.
- **Pointer Storage:** A global pointer saves access to the text, useful for sending or clearing.
- **Size and Hint:** It shows a placeholder ("Write Your Message") and supports multiple lines.
- **Character Limit:** The field limits the number of characters (e.g., 255).

3- Action Buttons

```
// Create Send Button
    lv_obj_t *send_btn = lv_btn_create(scr);
    lv_obj_set_size(send_btn, 85, 40);
    lv_obj_align(send_btn, LV_ALIGN_BOTTOM_LEFT,10,-170); // Align to the left
    lv_obj_add_event_cb(send_btn, send_text_message_event_cb,LV_EVENT_CLICKED,NULL);
    lv_obj_t *label2 = lv_label_create(send_btn);
    lv_label_set_text(label2, "Send");
    lv_obj_center(label2);
    // Create Voice Button
    lv_obj_t *voice_btn = lv_btn_create(scr);
    lv_obj_add_event_cb(voice_btn, record_button_event,LV_EVENT_CLICKED,NULL);
    lv_obj_t *label3 = lv_label_create(voice_btn);
    lv_label_set_text(label3, "Voice");
    lv_obj_center(label3);
    // Create back Button
    lv_obj_t *back_btn = lv_btn_create(scr);
    lv_obj_add_event_cb(back_btn, back_to_UI_1_event_cb,LV_EVENT_CLICKED,NULL);
    lv_obj_t *label4 = lv_label_create(back_btn);
    lv_label_set_text(label4, "<");
    lv_obj_center(label4);
```

Code Snippet 5-16:Action Buttons.

- **Send Button:** Sends the typed message using a send_text_message_event_cb function. It's placed at the bottom right.
- **Voice Button:** Starts a voice recording using record_button_event, next to the send button.
- **Back Button:** Takes the user back to the main chat screen (UI_1), placed at the top left.

5.7.3 Playing Voice Messages

When a voice message is received, we show a bubble with a play button. The user can press the play icon to hear the message through the speaker.

Implementation

The create_voice_message_bubble function builds the visual layout for voice messages in the main chat screen (UI_1). It not only displays the message but also includes a play button to let users listen to the audio easily. The design clearly distinguishes audio messages from text and shows whether they were sent or received.

Sent vs. Received Styling

```c
static void create_voice_message_bubble(lv_obj_t *parent, const char *file_path, bool is_sent) {
    lv_obj_t *msg_bubble = lv_obj_create(parent);
    lv_obj_remove_flag(msg_bubble, LV_OBJ_FLAG_SCROLLABLE);

    // Voice messages are slightly narrower than text messages.
    lv_obj_set_width(msg_bubble, LV_PCT(80));
    lv_obj_set_height(msg_bubble, LV_SIZE_CONTENT);
    lv_obj_set_style_pad_all(msg_bubble, 5, LV_PART_MAIN);
    lv_obj_set_style_radius(msg_bubble, 15, LV_PART_MAIN);

    // Apply specific styles based on whether the message was sent or received.
    if (is_sent) {
        lv_obj_set_align(msg_bubble, LV_ALIGN_TOP_RIGHT);
        lv_obj_set_style_bg_color(msg_bubble, lv_color_hex(0x007bff), LV_PART_MAIN);
        lv_obj_set_style_bg_opa(msg_bubble, LV_OPA_100, LV_PART_MAIN);
        lv_obj_set_style_text_color(msg_bubble, lv_color_white(), LV_PART_MAIN);
        lv_obj_set_style_pad_right(msg_bubble, 10, LV_PART_MAIN);
    } else {
        lv_obj_set_align(msg_bubble, LV_ALIGN_TOP_LEFT);
        lv_obj_set_style_bg_color(msg_bubble, lv_color_hex(0xE0E0E0), LV_PART_MAIN);
        lv_obj_set_style_bg_opa(msg_bubble, LV_OPA_100, LV_PART_MAIN);
        lv_obj_set_style_text_color(msg_bubble, lv_color_white(), LV_PART_MAIN);
        lv_obj_set_style_pad_left(msg_bubble, 10, LV_PART_MAIN);
    }
```

Code Snippet 5-17: Sending vs Receiving style.

- **Alignment**:
  - Sent messages are shown on the right.
  - Received messages appear on the left.
- **Color**:
  - Sent = blue with white text.
  - Received = gray with dark text.
- **Padding**:
  - Small side padding prevents edge crowding.

Play Button

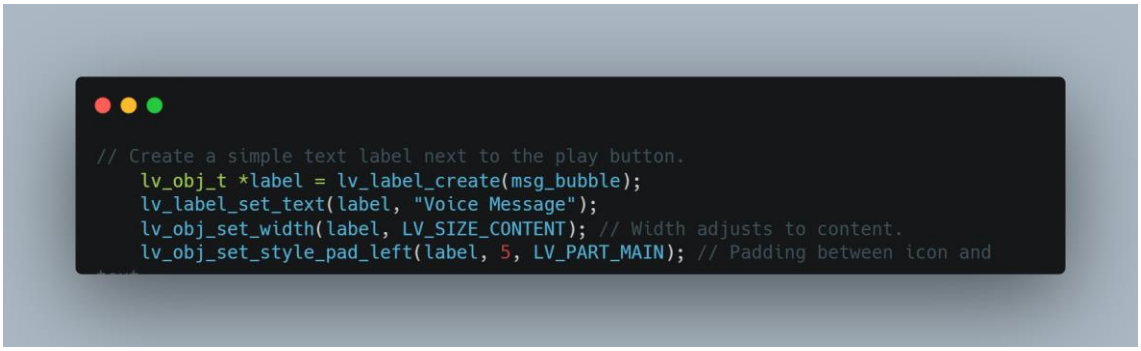
```
// Create the Play Button itself.
    lv_obj_t *play_btn = lv_btn_create(msg_bubble);
    lv_obj_set_size(play_btn, 40, 40); // Fixed size for the round button.
    lv_obj_set_style_radius(play_btn, LV_RADIUS_CIRCLE, LV_PART_MAIN); // Make it perfectly round.
    lv_obj_set_style_bg_color(play_btn, lv_color_hex(0x28a745), LV_PART_MAIN); // Green color for play.
    lv_obj_set_style_bg_opa(play_btn, LV_OPA_100, LV_PART_MAIN);
    lv_obj_set_style_border_width(play_btn, 0, LV_PART_MAIN); // No border.
    lv_obj_set_style_pad_all(play_btn, 0, LV_PART_MAIN); // No internal padding for the button itself.

    // Create a label inside the button for the play icon.
    lv_obj_t *play_icon = lv_label_create(play_btn);
    lv_label_set_text(play_icon, LV_SYMBOL_PLAY); // Use LVGL's built-in play symbol.
    lv_obj_set_style_text_color(play_icon, lv_color_white(), LV_PART_MAIN); // Corrected:
lv_color_white()
    lv_obj_center(play_icon); // Center the icon within the button.
```

Code Snippet 5-18: Create voice play button.

- **Button Design**:
  - A green, round button is added to play the audio.
  - A triangle icon (▶) is placed inside to show it's for playback.
- **Event Handling**:
  - When clicked, it runs voice_message_event_cb and plays the correct audio file using the attached file path.

Voice Message Label



```
// Create a simple text label next to the play button.
    lv_obj_t *label = lv_label_create(msg_bubble);
    lv_label_set_text(label, "Voice Message");
    lv_obj_set_width(label, LV_SIZE_CONTENT); // Width adjusts to content.
    lv_obj_set_style_pad_left(label, 5, LV_PART_MAIN); // Padding between icon and
```

Code Snippet 5-19: Create voice message label.

- **Label Text**: Displays “Voice Message” next to the play button as a hint.
- **Style**: Color and spacing are adjusted to match the message type and improve readability.

Basic Structure and Style

- **Bubble Creation**: A new container is created to hold the voice message.
- **Fixed Size**: The bubble is narrower than text messages and adjusts its height based on content.
- **Non-Scrollable**: Scrolling is disabled for individual bubbles.
- **Rounded Corners**: Smooth edges are added for a modern look.
- **Padding**: Space is added inside the bubble for clean spacing.

Chapter Six

6 SYSTEM INTEGRATION

6.1 INTRODUCTION

This chapter presents the integration phase of our secure communication system, built using Raspberry Pi 3 devices. Following the successful development of individual system modules (including the communication protocol, encryption mechanism, hardware setup, and user interface) this phase focuses on assembling all components into a fully operational prototype.

Our goal in this here is to demonstrate how the messaging engine, audio processing, graphical interface (LVGL), and network communication modules are connected to form a cohesive and reliable system. Each Raspberry Pi functions as a peer node, capable of sending and receiving both text and voice messages through a wireless transceiver network. All data transmissions are protected using a Hybrid encryption scheme developed in earlier phases.

Given the target use case of IoT experimentation, the integration was designed with modularity, efficiency, and offline operability in mind. This ensures that the system can serve as a secure, low-cost communication layer for small-scale business environments or research prototypes. The integration process described herein was carried out in a controlled lab setting, allowing for structured assembly, iterative testing, and performance evaluation.

The following sections document the system architecture, the LVGL-based UI integration, the unified message handling logic, encryption flow coordination, and system-level synchronization. Challenges encountered during integration and the strategies used to overcome them are also discussed, providing a comprehensive view of the project's assembly phase.

6.2 SOFTWARE INTEGRATION ARCHITECTURE

The integration of this system brings together four key software modules: **User Interface**, **Text Messaging**, **Audio Processing**, and **Network Communication**. Each module was developed independently in C, organized into separate .c and .h source and header files, and then assembled into a unified system where the UI drives most of the functionality via event-triggered callbacks.

The design follows a layered architecture, with clear boundaries between presentation, logic, and hardware abstraction.

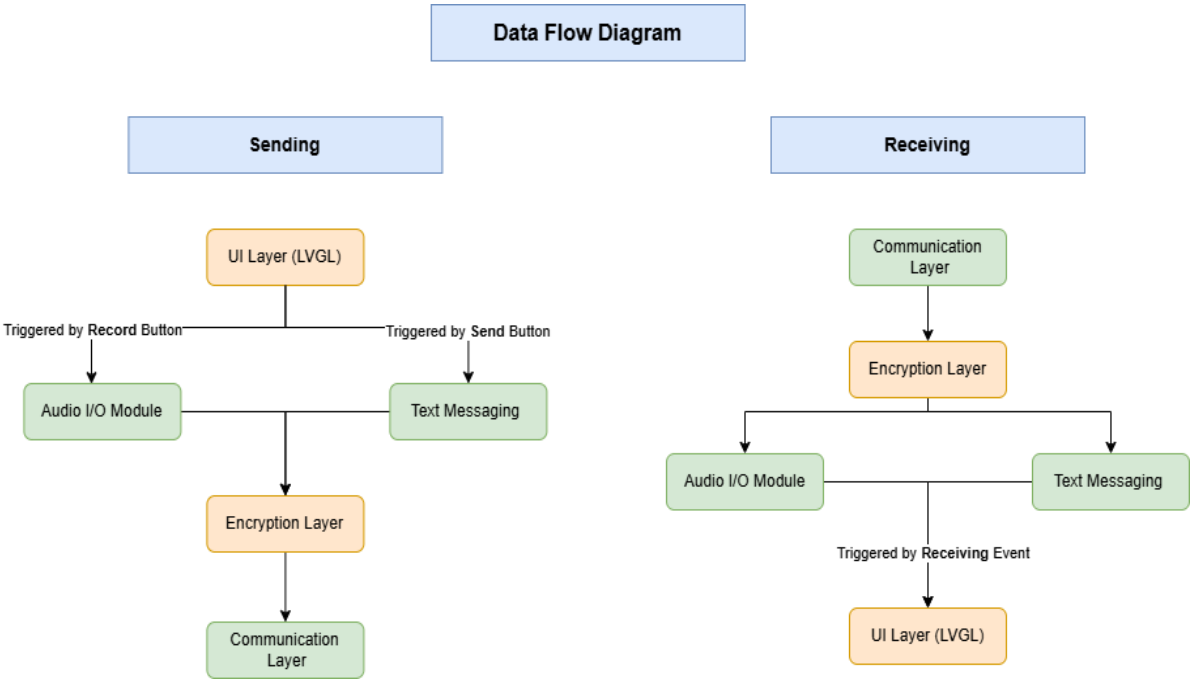


Figure 6-1: Data Flow Diagram.

This diagram illustrates the bidirectional flow of data between modules in the system during message transmission and reception. On the **sending side**, user interactions through the LVGL-based UI trigger either text message creation or audio recording. These are then passed through the encryption layer before being transmitted via the network communication layer.

On the **receiving side**, incoming messages are decrypted, classified as text or audio, and routed to the appropriate playback or display component through the UI. This design reflects a modular event-driven architecture where the UI initiates all operations and each functional layer performs a specific, isolated task.

6.2.1.1 Module Overview

Table 6-1: Software Modules and Responsibilities

Module	Role
touch.c/h	Interfaces with the touchscreen hardware via **LVGL** driver
tranciver.c/h	Handles sending and receiving data through the wireless transceiver using node ID addressing
audio.c/h	Records and plays **.wav** audio files using the microphone and speaker
encryption.c/h	Performs encryption and decryption for both text and audio messages, and manages session key operations
message.c/h	Structures messages, applies integrity hashing, and coordinates with the encryption module

6.3 BUILD SYSTEM INTEGRATION USING MAKEFILE

6.3.1 Introduction to Makefile

In software development, particularly in C-based projects, managing the build process becomes increasingly complex as the project grows. A Makefile provides an effective solution by automating compilation and linking tasks. It defines a set of rules that determine how to build different components of the project, ensuring consistency and reducing manual effort.

Using a Makefile allows for:

- Incremental compilation: only modified files are rebuilt, significantly improving build times.
- Dependency management: changes in source files trigger only necessary recompilation.

- Custom build workflows: developers can define targets like `all`, `clean`, `run`, or `install`.
- Toolchain and flag control: compile and link flags can be precisely specified per target.

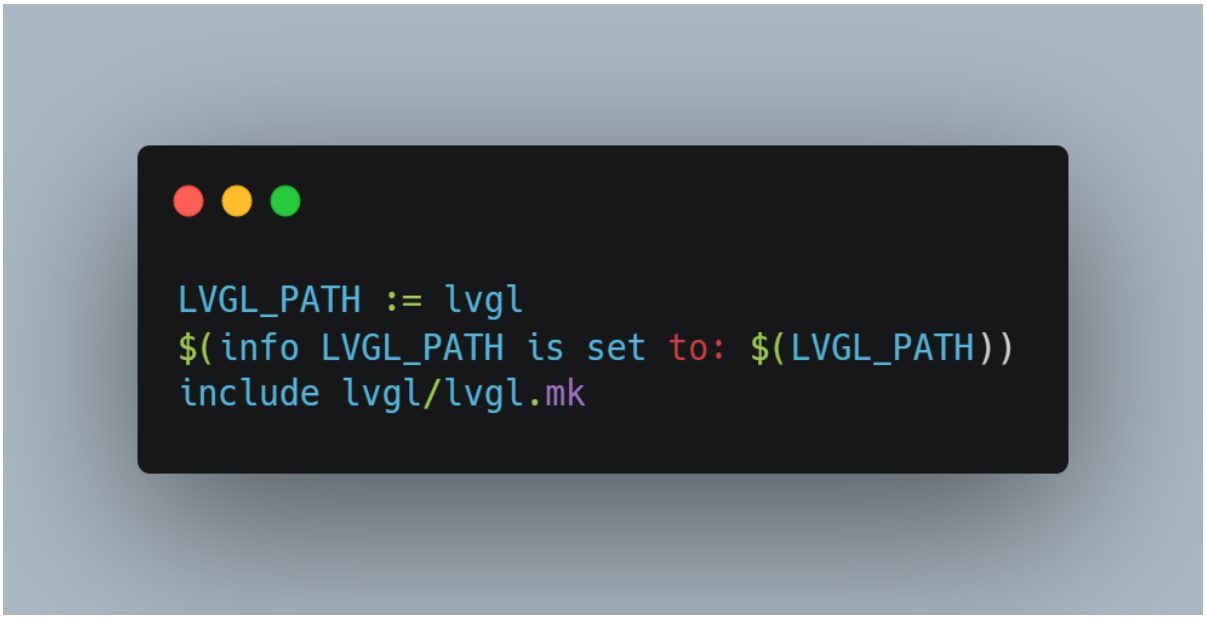
Make is widely supported and highly adaptable, making it especially useful in embedded systems development where fine-grained control over the compilation process is often required.

6.3.2 LVGL Integration with Makefile

LVGL (Light and Versatile Graphics Library) is an open-source embedded GUI library that supports Makefile-based integration. LVGL simplifies its use in C projects by providing a dedicated `lvgl.mk` file that handles its own compilation rules.

To integrate LVGL into the build process, a developer needs to:

1. Clone or download the LVGL source code.
2. Include the provided `lvgl.mk` in the main Makefile.
3. Define the `LVGL_PATH` variable pointing to the LVGL directory.
4. Ensure include paths are correctly set for the compiler.



Code Snippet 6-1: LVGL Integration with Makefile

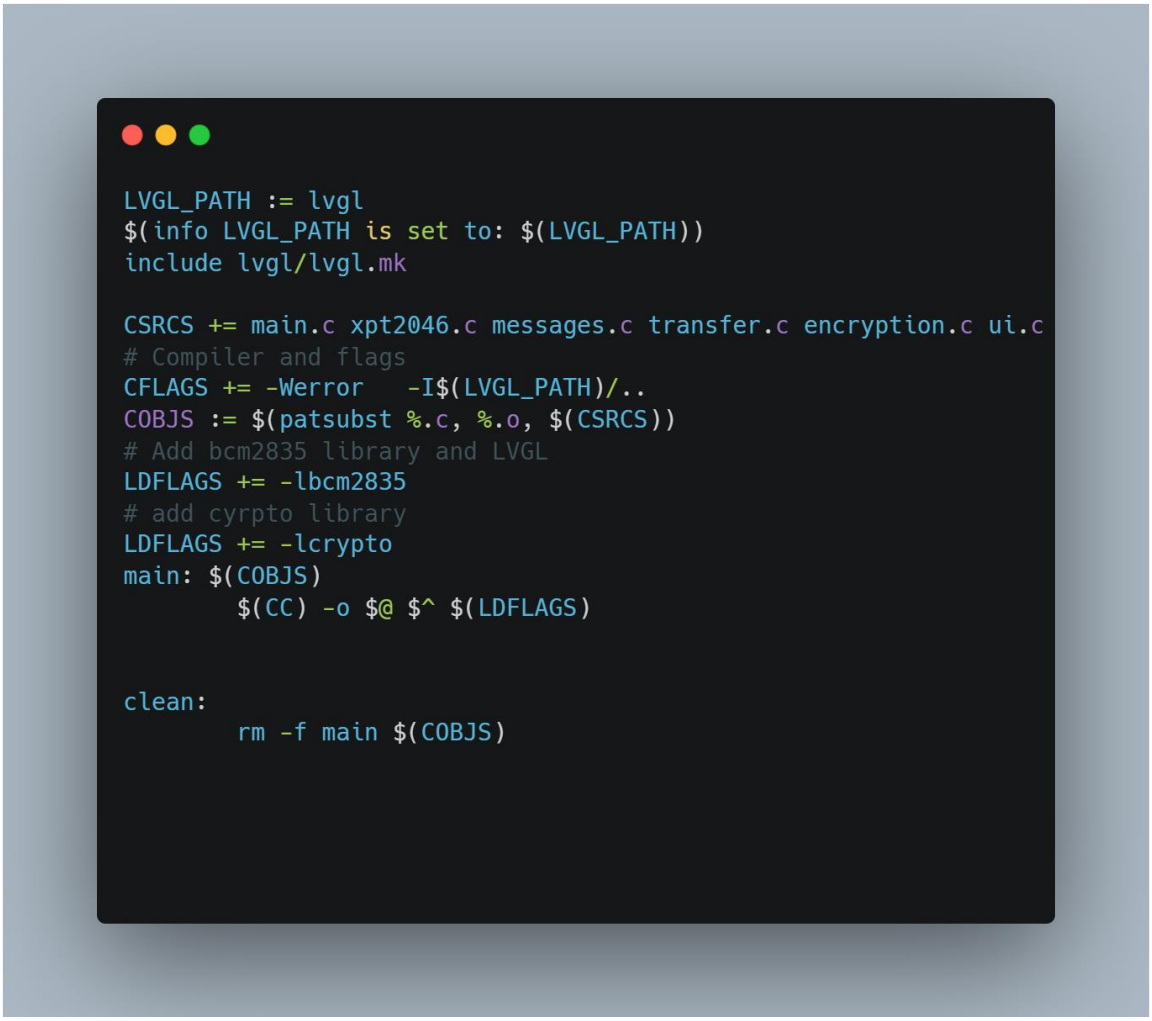
This structure enables LVGL to be compiled alongside user code without additional configuration.

6.3.3 Project-Specific Makefile Implementation

This project was developed in the C programming language and designed to run on Raspberry Pi hardware. It interfaces with GPIO and SPI through the `bcm2835` library and utilizes the OpenSSL `crypto` library for encryption functions. Additionally, it features a touch interface and display system built using LVGL.

The Makefile automates the following tasks:

- Compiles all C source files from the project and LVGL.
- Links against required libraries (`bcm2835`, `crypto`).
- Produces a single executable named `main`.



```
LVGL_PATH := lvgl
$(info LVGL_PATH is set to: $(LVGL_PATH))
include lvgl/lvgl.mk

CSRCS += main.c xpt2046.c messages.c transfer.c encryption.c ui.c
# Compiler and flags
CFLAGS += -Werror   -I$(LVGL_PATH)/..
COBJS := $(patsubst %.c, %.o, $(CSRCS))
# Add bcm2835 library and LVGL
LDFLAGS += -lbcm2835
# add cyrpto library
LDFLAGS += -lcrypto
main: $(COBJS)
        $(CC) -o $@ $^ $(LDFLAGS)


clean:
        rm -f main $(COBJS)
```

Code Snippet 6-2: whole make file code

This Makefile setup allows for a straightforward and efficient build process. The `lvgl.mk` handles LVGL-specific compilation, while the rest of the logic compiles user code and links required system libraries.

6.3.4 Build and Clean Instructions

To build the project, simply run the following command in the terminal from the project root directory: ***make***

This will compile all source and LVGL files and generate the `main` executable.

To remove all compiled files and reset the build environment, run: ***make clean***

This ensures that future builds start from a clean state.

Using a Makefile for this project provided a robust, lightweight, and fully transparent build system. It allowed full control over compilation flags, external dependencies, and build structure.

By incorporating LVGL through its native Makefile support, and by linking necessary libraries for hardware and cryptographic operations, the resulting system was both modular.

This setup is especially appropriate for embedded Linux environments like the Raspberry Pi, where simple and deterministic build logic is often preferred.

Chapter Seven

7 REFERENCE

Books:

[1] tallings, W. (2017). Cryptography and Network Security: Principles and Practice. Pearson. [online]. Available: https://www.uoitc.edu.iq

Reports and Handbooks:

[2] TCP/IP Sockets in C: Practical Guide for Programmers. [online]. Available: https://cs.baylor.edu

[3] FIPS 119 Advanced Encryption Standard (AES). [online]. Available: https://nvlpubs.nist.gov

Standards and Protocols:

[4] IEEE 802.11p Standard. [Online]. Available: http://www.ieee.org

[5] RFC: 793 TRANSMISSION CONTROL PROTOCOL. [Online]. Available: https://www.ietf.org

Technologies and Components:

[6] Raspberry Pi Official Website. [Online]. Available: http://www.raspberrypi.org

[7] Raspberry Pi Documentation. [Online]. Available: http://www.raspberrypi.org/documentation

[8] ILI9341 data sheet. [Online]. Available: https://www.alldatasheet.com

[9] nRF24L01+ data sheet. [Online]. Available: https://cdn.sparkfun.com

[10] BCM2835: C library for Broadcom. [Online]. Available: https://www.airspayce.com

[11] LVGL documentation. [Online]. Available: https://docs.lvgl.io

[12] OpenSSL documentation. [Online]. Available: https://docs.openssl.org