**Fayoum University**

**Engineering Faculty**

**Electrical Engineering Department**

**B. Eng. Final Year Project**

**Secure Communication Network**

**By:**

**Ahmed Abdulla Mourad**

**Ahmed Mohamed Eid**

**Mohamed Shaban El-sayed**

**Moaz Mohamed Radwan**

**Supervised By:**

**Dr: Mohamed Hamdy**

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.

[1. Project Overview 7](#_Toc201635604)

[2. Hardware Components 7](#_Toc201635605)

[3. RASPBERRY PI (THE MAIN ECU) 7](#_Toc201635606)

[**INTRODUCTION :** 7](#_Toc201635607)

[3.2 TOUCH SCREEN 20](#_Toc201635608)

[3.3 MICK 37](#_Toc201635609)

[3.4 SPEAKER 37](#_Toc201635610)

[3.5 TRANCEIVER 37](#_Toc201635611)

[3.5.1 Introduction to nRF24L01 37](#_Toc201635612)

[3.5.2 Key Features: 37](#_Toc201635613)

[3.5.3 Peer-to-Peer Communication Explained 38](#_Toc201635614)

[3.5.4Hardware Connections (Raspberry Pi 3 Example) 38](#_Toc201635615)

[3.5.5 Software Setup (C Programming with RF24 Library) 40](#_Toc201635616)

[3.5.6 Communication Pipes and Addressing in Detail 40](#_Toc201635617)

[3.5.6 Troubleshooting Common Issues 42](#_Toc201635618)

[3.Communication and Tcp 43](#_Toc201635619)

[3.1 Socket Programming in C 43](#_Toc201635620)

[3.1.1 Introduction Socket programming 43](#_Toc201635621)

[3.1.2 Components of Socket Programming 43](#_Toc201635622)

[3.2 Introduction to TCP 44](#_Toc201635623)

[3.3 Properties of TCP 44](#_Toc201635624)

[3.4How Does TCP Work 45](#_Toc201635625)

[3.5 Understanding TCP through visualization 48](#_Toc201635626)

[3.6 Hardware Setup (Raspberry Pi 3) 58](#_Toc201635627)

[**3.7 Software Setup** 59](#_Toc201635628)

[3.8 Disadvantages of TCP 61](#_Toc201635629)

[4. 4.Encryption and Security Protocols 61](#_Toc201635630)

[4.1 Introduction to Cryptographic Principles 61](#_Toc201635631)

[4.2 System Security Requirements 64](#_Toc201635632)

[4.3 Hybrid Encryption Design 67](#_Toc201635633)

[4.4 Implementation of Hybrid Encryption with OpenSSL 70](#_Toc201635634)

[4.5 Integration with Other Modules 73](#_Toc201635635)

[4.6 Summery 74](#_Toc201635636)

[5. 75](#_Toc201635637)

[6. System Integration 75](#_Toc201635638)

[1.1 Introduction 75](#_Toc201635639)

[1.2 Software Integration Architecture 76](#_Toc201635640)

[1.2.1 Module Overview 77](#_Toc201635641)

[Build System Integration Using Makefile 78](#_Toc201635642)

[Introduction to Makefile 78](#_Toc201635643)

[LVGL Integration with Makefile 78](#_Toc201635644)

[Project-Specific Makefile Implementation 79](#_Toc201635645)

[Build and Clean Instructions 80](#_Toc201635646)

[7. References 81](#_Toc201635647)

[Figure ‎2‑1: Threat model highlighting attack vectors. 49](#_Toc201397758)

[Figure ‎2‑2: Encryption Workflow. 51](#_Toc201397759)

[Figure ‎2‑3: Comparison of encryption approaches. 52](#_Toc201397760)

[Figure ‎2‑4: Hybrid encryption workflow 54](#_Toc201397761)

[Figure ‎2‑5: Generation of key pair (public, private) 56](#_Toc201397762)

[Figure ‎2‑6: Generate Session Key 56](#_Toc201397763)

[Figure ‎2‑7: RSA Encryption 56](#_Toc201397764)

[Figure ‎2‑8: AES Encryption 57](#_Toc201397765)

[Figure ‎3‑1: System Data Flow Diagram. 61](#_Toc201397766)

[Figure ‎3‑2: LVGL Integration with Makefile 63](#_Toc201397767)

[Table ‎4‑1: Algorithm and Libraries 51](#_Toc201397992)

[Table ‎5‑1: Software Modules and Responsibilities 62](#_Toc201397993)

[Code Snipped ‎3‑1: whole make file code 64](#_Toc201397975)

# Project Overview

# Hardware Components

# RASPBERRY PI (THE MAIN ECU)

## **INTRODUCTION :**

In this chapter, 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.

* + 1. **Raspberry Pi 4 in Our Project:**

For our project, we utilized the Raspberry Pi 4, which offers a significant improvement in speed and performance compared to its predecessors. The Raspberry Pi 4 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.

**3.1.3 Key Features of Raspberry Pi 4:**

1. Silent and Energy-Efficient:

The Raspberry Pi 4 operates silently without a fan and consumes significantly less power than traditional computers.

1. Fast Networking:

Equipped with Gigabit Ethernet, onboard wireless networking, and Bluetooth, the Raspberry Pi 4 ensures fast and reliable connectivity.

1. SPI communication protocol

SPI communication protocol is a must to connect the screen with raspberry pi and transceiver with raspberry pi also

1. 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.

1. Choice of RAM:

The Raspberry Pi 4 is available in different configurations, offering 1GB, 2GB, 4GB, or 8GB of RAM to meet various performance needs. These features make the Raspberry Pi 4 an excellent choice for a wide range of applications, from simple computing tasks to more complex projects involving robotics and automation.

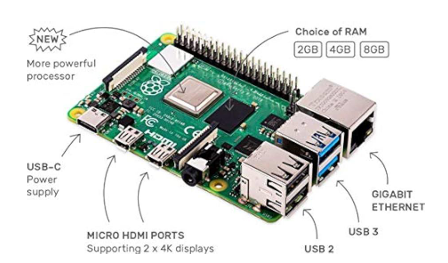


Figure 3‑1-1: Raspberry Pi ECU

* + 1. **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:

1. Cost-Effective:

The Raspberry Pi is an affordable computer, making it accessible for beginners and ideal for projects with budget constraints.

1. 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.

1. Versatility:

Capable of running various operating systems and applications, the Raspberry Pi is suitable for a broad range of projects.

1. GPIO Pins:

The General-Purpose Input/Output (GPIO) pins make it easy to connect a wide variety of devices, facilitating custom electronics projects.

1. Community Support:

A large, active community of users and developers shares knowledge and resources, making it easy to find help and solutions to problems.

1. 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.

* + 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. 18

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 19

• Features: Improved network speeds and thermal performance, enhanced power management.

8. 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.

9. 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.

10. 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 3-2 Provides a summary of Raspberry Pi versions timeline.

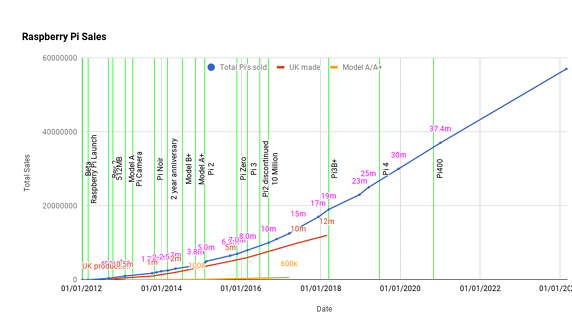


Figure 3‑1-2: Raspberry Pi versions timeline

**3.1.5 RASPBERRY PI HARDWARE**

Overview:

In this section, we will discuss the hardware aspects of the Raspberry Pi 4, 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.

Hardware Components of Raspberry Pi

The hardware components of the Raspberry Pi include:

1. CPU/GPU:

The ARM Cortex-A72 CPU and VideoCore VI GPU provide the processing power and graphical capabilities.

1. RAM:

Depending on the model, the Raspberry Pi 4 can have 2GB, 4GB, or 8GB of RAM.

Networking:

Integrated Gigabit Ethernet, Wi-Fi, and Bluetooth enable robust connectivity options. 21

1. 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:

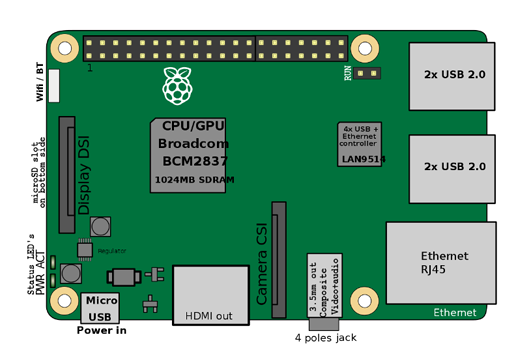
The USB-C power supply ensures sufficient power for all components and peripherals.

Figure 3‑1-3: 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.

**3.1.4 Raspberry Pi 4 Specifications**

The Raspberry Pi 4 represents a significant upgrade over previous models, offering enhanced performance and connectivity options. Here are some key specifications:

1. Processor:

1.5 GHz quad-core ARM Cortex-A72

1. RAM:

Options of 2GB, 4GB, and 8GB

1. Networking:

Gigabit Ethernet, Wi-Fi 802.11ac, Bluetooth 5.0 22

USB Ports:

2 USB 3.0 ports and 2 USB 2.0 ports

1. Video Output:

2 micro-HDMI ports supporting up to 4K resolution

1. Storage:

MicroSD card slot for storage and operating system

1. Power:

USB-C power supply

**3.1.6 Layout and Interfacing of GPIO Pins**

The pin diagram of the Raspberry Pi 4 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 4 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.

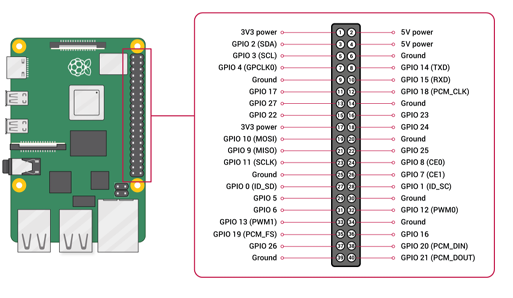
• Special Function Pins: Dedicated pins for communication protocols like I2C, SPI, and UART.

Figure3-1-4: Raspberry Pi Pin Diagram

**3.1.6 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

1. 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.

1. 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 your C drivers and building your project.

1. 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.

1. 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.

## TOUCH SCREEN

**(a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color)**

Figure 3-2-1:touch screen

**3.2.1 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.

**3.2.2 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:

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.

1. 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.

1. 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.

1. 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.

1. Detailed MCU and RGB Interface Functionality

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

**3.2.3 MCU Interfaces**

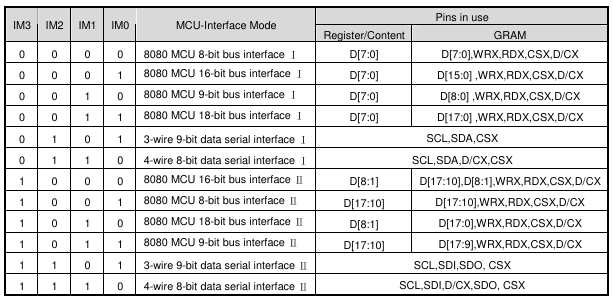
The driver offers both parallel and serial interfaces to communicate with a host MCU. The selection between these modes and their specific configurations is determined by the external IM[3:0] pins.

Figure 3-2-2:interface table

1. 8080-I and 8080-II Series Parallel Interfaces

The ILI9341 supports 8-, 9-, 16-, and 18-bit data bus configurations for both 8080-I and 8080-II series MCUs.

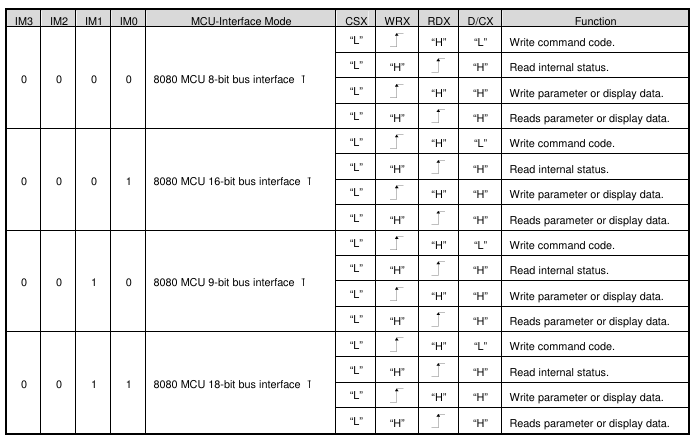


Figure 3-2-3: 8080 Ⅰ series parallel interface table

* **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.

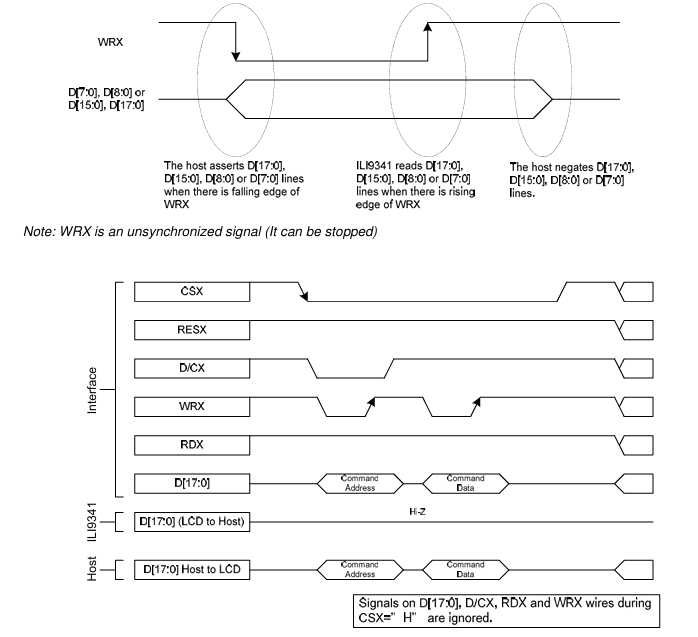


Figure 3-2-4:write cycle for the 8080 Ⅰ

* **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.

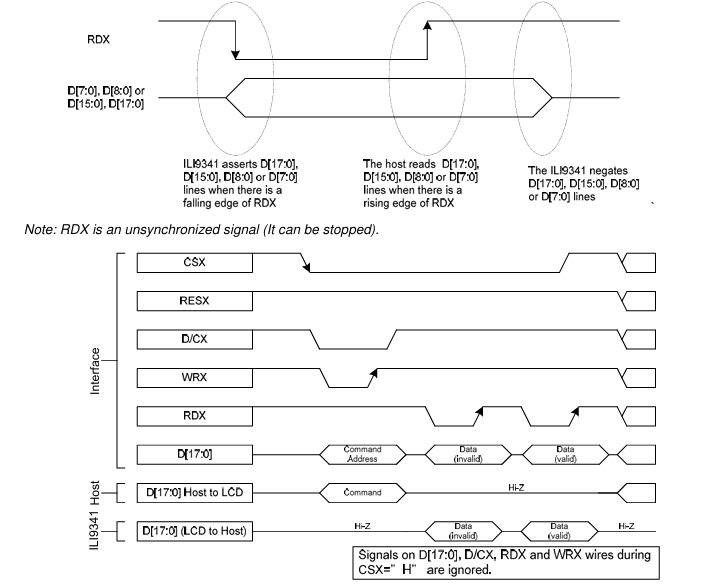


Figure 3-2-5: read cycle for the 8080 Ⅰ

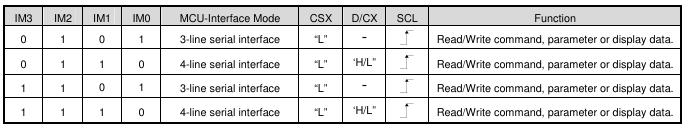
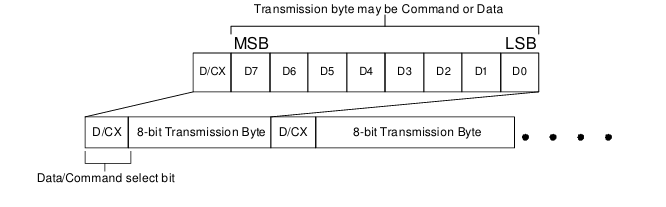
1. Serial Interface (SPI)

Figure 3-2-6: series interface table

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 transmission byte.

Figure 3-2-7: 3-line SPI sequence 

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

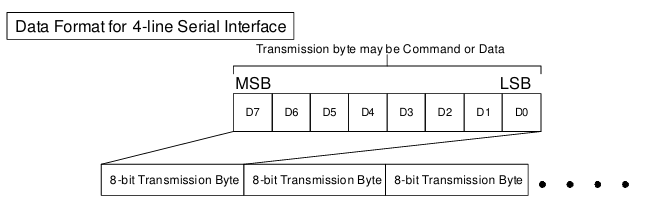


Figure 3-2-8: 4-line SPI sequence

* **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.

**3.2.3 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.

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.

1. 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.

1. 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.

1. 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.

**3.2.4 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.

**3.2.5 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.

**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.

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.

1. 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).

**3.2.7 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.

* **Power Supply Order**: VDDI and VCI can be applied and powered down in any order.
* **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.
* **Power Off**: Different power-down timings apply depending on whether the LCD is in Sleep Out or Sleep In mode.
* **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.
* **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.
* **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.

**3.2.8 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.

1. **Partial Mode On, Idle Mode Off, Sleep Out**:

Partial display area, 262,144 colors.

1. **Normal Mode On (Full Display), Idle Mode On, Sleep Out**:

Full display area, 8 colors.

1. **Partial Mode On, Idle Mode On, Sleep Out**:

Partial display area, 8 colors.

1. **Sleep In Mode**:

DC/DC converter, internal oscillator, and panel driver stopped. MCU interface and memory remain active, preserving memory contents.

1. **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.

1. **Power Off Mode**:

Both VCI and VDDI removed

## 3.3 MICK

## 3.4 SPEAKER

## 3.5 TRANCEIVER

Peer-to-Peer Communication with nRF24L01 Modules

#### 3.5.1 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.

#### 3.5.2 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.

#### 3.5.3 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.

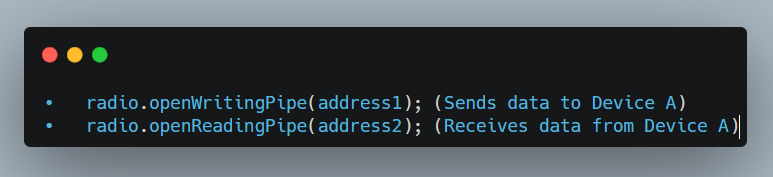
Similarly for Device B:

* 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).

**Device A's Configuration:**

**Device B's Configuration:**



#### 3.5.4Hardware 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

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) |

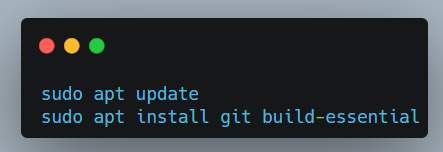
Connection to Raspberry Pi 3 (Both Devices Identically):

|  |  |  |
| --- | --- | --- |
| 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 |

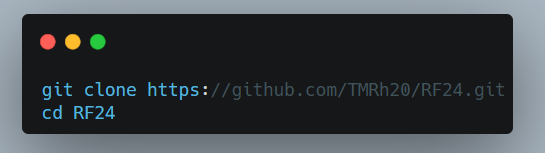
#### 3.5.5 Software Setup (C Programming with RF24 Library)

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:

Bash

3. Clone the RF24 Library:

Bash

4. Build and install the RF24 Library for Raspberry Pi

#### 3.5.6 Communication Pipes and Addressing in Detail

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

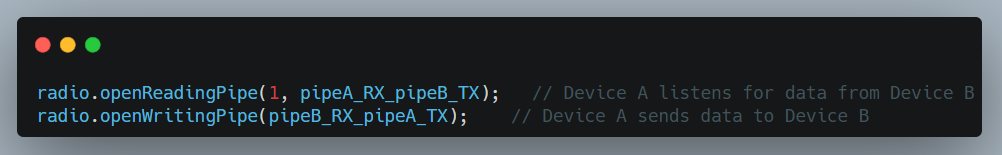
Each pipe has a unique address.

* **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.

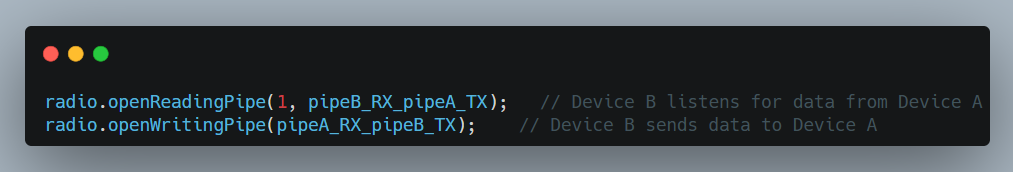
**Defining Addresses for Bidirectional Communication:**

* **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

**Device A Setup:**

****

**Device B Setup:**

 **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.

**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.

#### 3.5.6 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.

# 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

1. Sockets

[Sockets](https://www.geeksforgeeks.org/socket-in-computer-network/) 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](https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/)).
* UDP Socket (Datagram Socket): Provides connectionless communication, faster but unreliable (i.e., [UDP protocol](https://www.geeksforgeeks.org/user-datagram-protocol-udp/)).

## 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:**

 TCP ensures that data packets are delivered without loss or corruption. If a packet is lost, TCP requests retransmission.

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:**

 TCP reassembles packets into the correct order.

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



**3.Error Checking:**

 TCP checks data for errors and requests replacements for corrupted packets, enhancing reliability.

**4.Connection-Oriented:**

 TCP establishes a connection between sender and receiver before transmitting data.

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



## 3.4How 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) :**

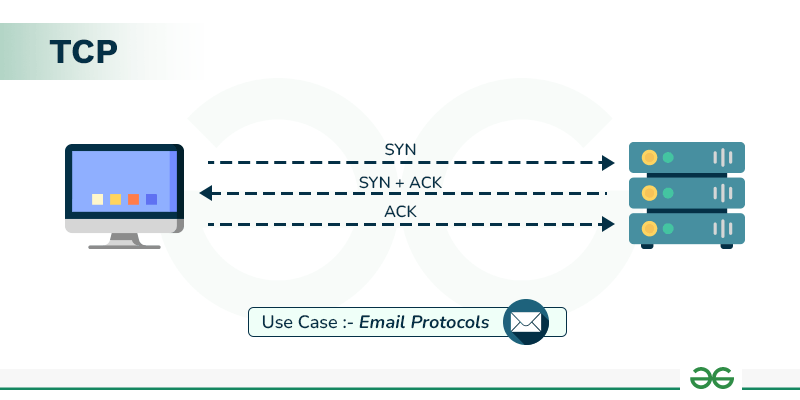
 The server responds with a TCP segment with both SYN and ACK flags set.

 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."

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.



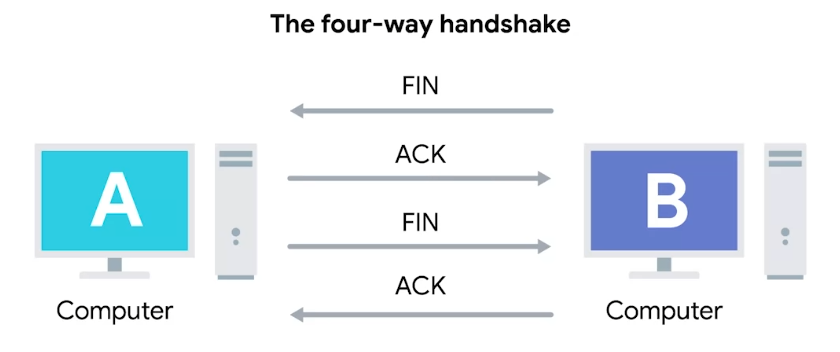
2.Acknowledgment:

 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.

## 3.5 Understanding TCP through visualization

**Term**

**Description**

**Server's Initial Sequence Number**

A random number, e.g., 2000

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

**Table ‎0‑1: TCP flags Table**

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.

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)



**1.Segment 1 – Client → Server:**

The client sends the first segment with a sequence number of 1001 ( initial sequence number + 1) .



**2.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) .



**3.Segment 2 – Client → Server:**

The client sends the second segment with a sequence number of 3001.



**4.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.

Connection Termination Example

**FIN – Client → Server:**

The client sends a FIN packet with a sequence number of 5001.



1. ACK for FIN – Server → Client:

The server acknowledges the client’s FIN packet with an ACK, setting the ACK number to 5002 ( 5001 + 1) .



2. FIN – Server → Client:

The server sends its own FIN packet with a sequence number of 5001.



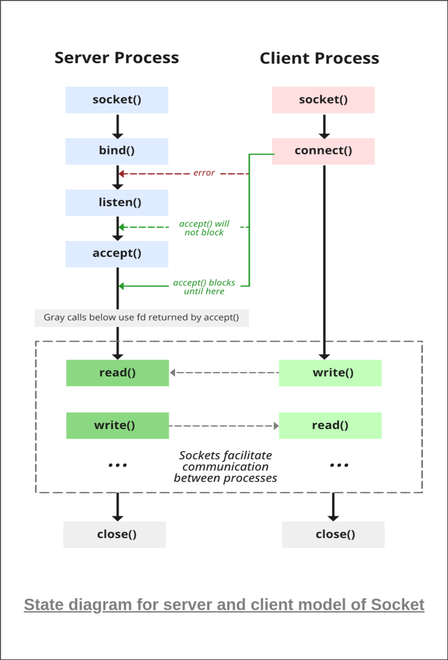
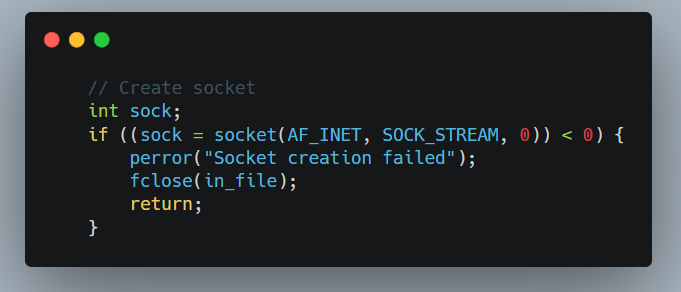
3. ACK for Server’s FIN – Client → Server:

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.

**2. Client-Server Model**

The [client-server model](https://www.geeksforgeeks.org/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 Model**Creating a Server-Side Process

The server is created using the following steps:

1.Socket Creation with socket() function

1.1 domain**(Communication Domain)**

Specifies the address family used for communication:

| **Domain** | **Description** |
| --- | --- |
| AF\_LOCAL | Communication between processes on the **same host** (formerly AF\_UNIX) |
| AF\_INET | IPv4 Internet protocols (most common for Internet communication) |
| AF\_INET6 | IPv6 Internet protocols |
| AF\_PACKET | Low-level packet interface (for direct network layer access) |

**1.2**type**(Socket Type)**

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.

1.3 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**

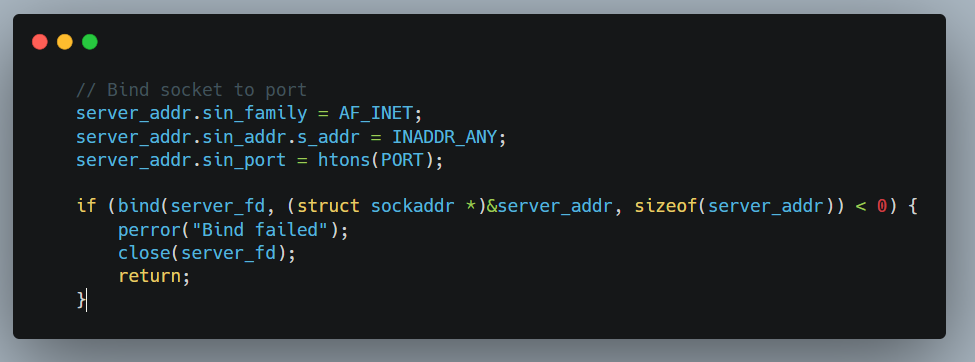
**1.4 Return Value (**sockfd**)**

* **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.

Full code for create a socket

2. **Bind**

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).

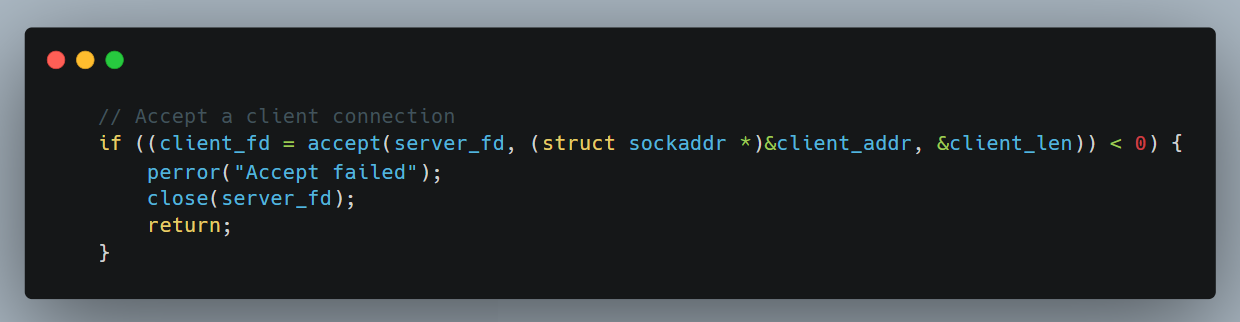
**Parameters**

| **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



**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.

**4.1 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:**

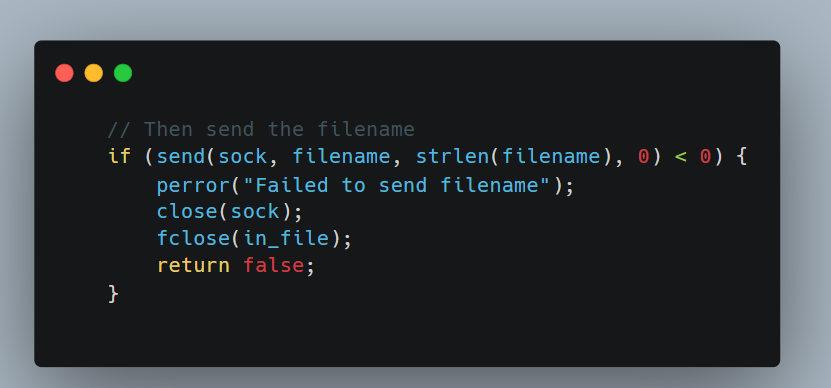
* **sockfd**: socket file descriptor returned by socket() and bind().
* **addr**: pointer to a struct sockaddr that will be hold the client's IP address and port number.
* **addrlen**: pointer to a variable that specifies the length of the address structure.

**5. Send/Recieve**

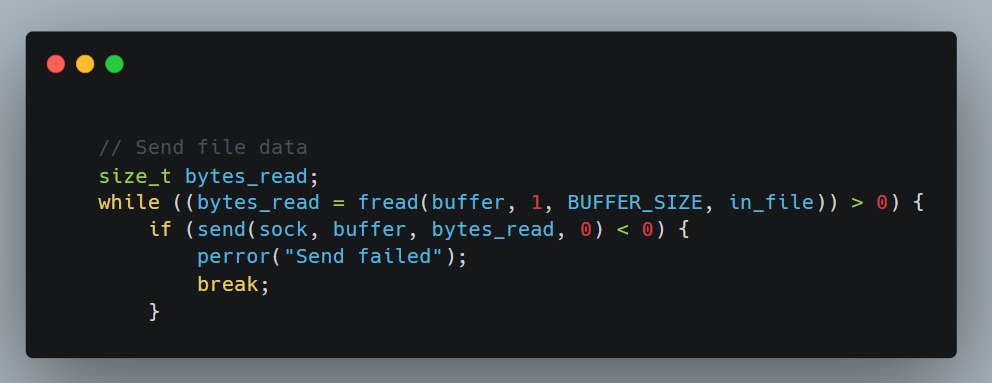
**For sending I will send file name first and then the data which will contain in a file name**

**or I can send string only without but it in file**

**first Send file name**

****

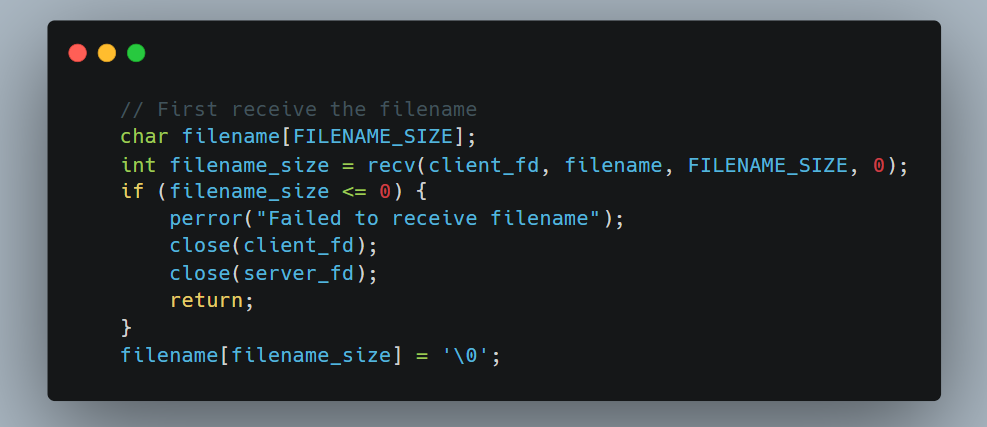
**Second send file data**

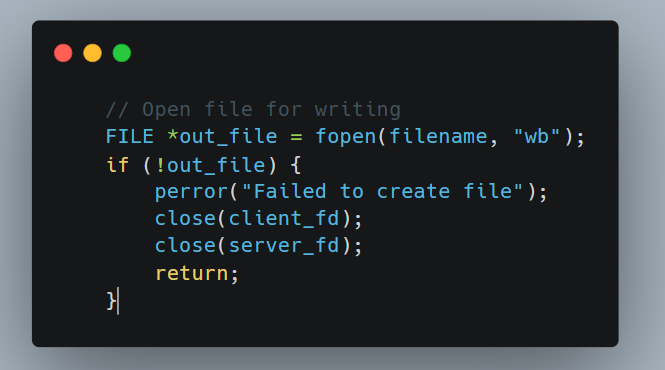
****

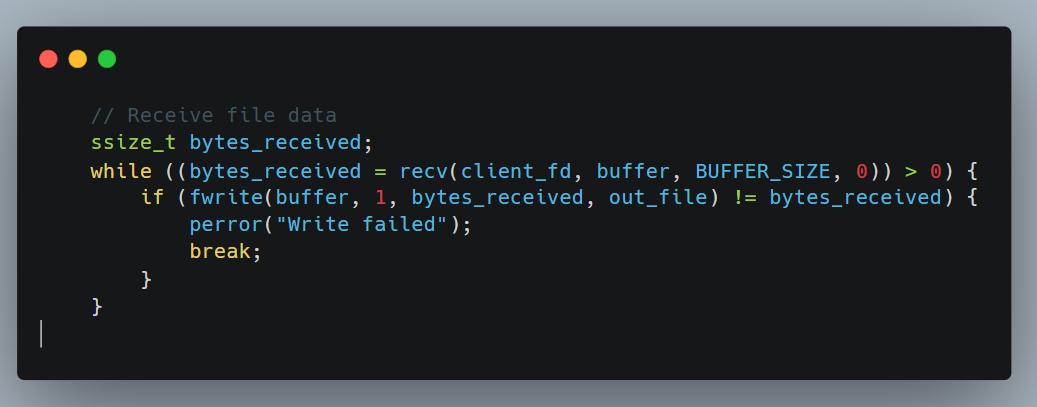
**For receiving I will receive file name first and then the data which will contain in a file**

**name or I can receive string only without but it in file**

**first receive file name**

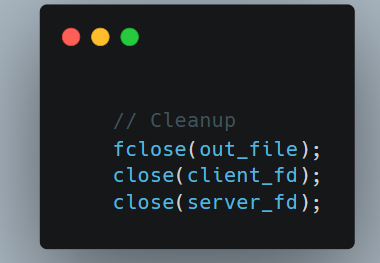
****

**Second Open file for writting**

**Third receive file data**

**6. Close**

Connection Termination ( Four-Way Handshake) :

****

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

**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**

Follow the below steps for creating a 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 are 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).

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) .

# 4.Encryption and Security Protocols

## 4.1 Introduction to Cryptographic Principles

Secure communication is a cornerstone of modern digital systems, particularly in networks where sensitive data is transmitted between nodes. Encryption ensures that even if messages are intercepted, their content remains confidential and unaltered. In this project, encryption safeguards both text and voice messages exchanged between Raspberry Pi (RPI) nodes from eavesdropping or tampering.

**The Need for Encryption**

Unsecured communication exposes systems to several risks:

* **Eavesdropping**: Attackers can intercept messages if transmitted in plaintext (Stallings, 2017).
* **Tampering**: Data modified in transit (e.g., altering a voice message) without detection (Katz & Lindell, 2020).
* **Spoofing**: Malicious actors impersonate legitimate nodes (Schneier, 2015).

Encryption addresses these threats by providing:

1. **Confidentiality**: Only authorized parties can read messages.
2. **Integrity**: Ensures messages are not modified in transit.
3. **Authentication**: Verifies the sender’s identity.

**Symmetric vs. Asymmetric Encryption**

Two fundamental encryption paradigms are employed in this project:

1. **AES (Advanced Encryption Standard) – Symmetric Encryption**
   1. **How it works**: Uses a single shared key for encryption and decryption.
   2. **Strengths**:
      1. Fast processing, ideal for encrypting large data (e.g., voice messages).
      2. Standardized by NIST, widely adopted (AES-256 is considered quantum-resistant for now).
   3. **Weakness**: Secure key distribution is challenging without a pre-shared key.
2. **RSA (Rivest-Shamir-Adleman) – Asymmetric Encryption**
   1. **How it works**: Uses a public key (shared openly) to encrypt and a private key (kept secret) to decrypt.
   2. **Strengths**:
      1. Solves key distribution problems (no pre-shared key needed).
      2. Enables digital signatures for authentication.
   3. **Weakness**: Computationally expensive; unsuitable for bulk data encryption.

**Hybrid Approach**: This project combines AES and RSA to leverage their strengths:

* **RSA** secures the exchange of AES session keys.
* **AES** encrypts the actual message payloads (text/voice).

## 4.2 System Security Requirements

A secure communication system must address both passive and active threats. This section defines the threat model, security goals, and design requirements tailored to the Raspberry Pi network.

**Threat Model**

The system assumes the following adversarial scenarios:

1. **Eavesdropping (Passive Attack)**:
   1. An attacker intercepts unencrypted messages transmitted between nodes.
   2. *Mitigation*: AES-256 encryption for message confidentiality.
2. **Man-in-the-Middle (MITM) Attack (Active Attack)**:
   1. An attacker alters or injects messages during transmission.
   2. *Mitigation*: RSA-based digital signatures to authenticate nodes.
3. **Replay Attack**:
   1. An attacker resends a valid message to disrupt the system (e.g., repeating a voice command).
   2. *Mitigation*: Timestamping messages and using session-specific nonces.
4. **Denial-of-Service (DoS)**:
   1. Overloading nodes with fake requests.
   2. *Mitigation*: Rate-limiting message processing.

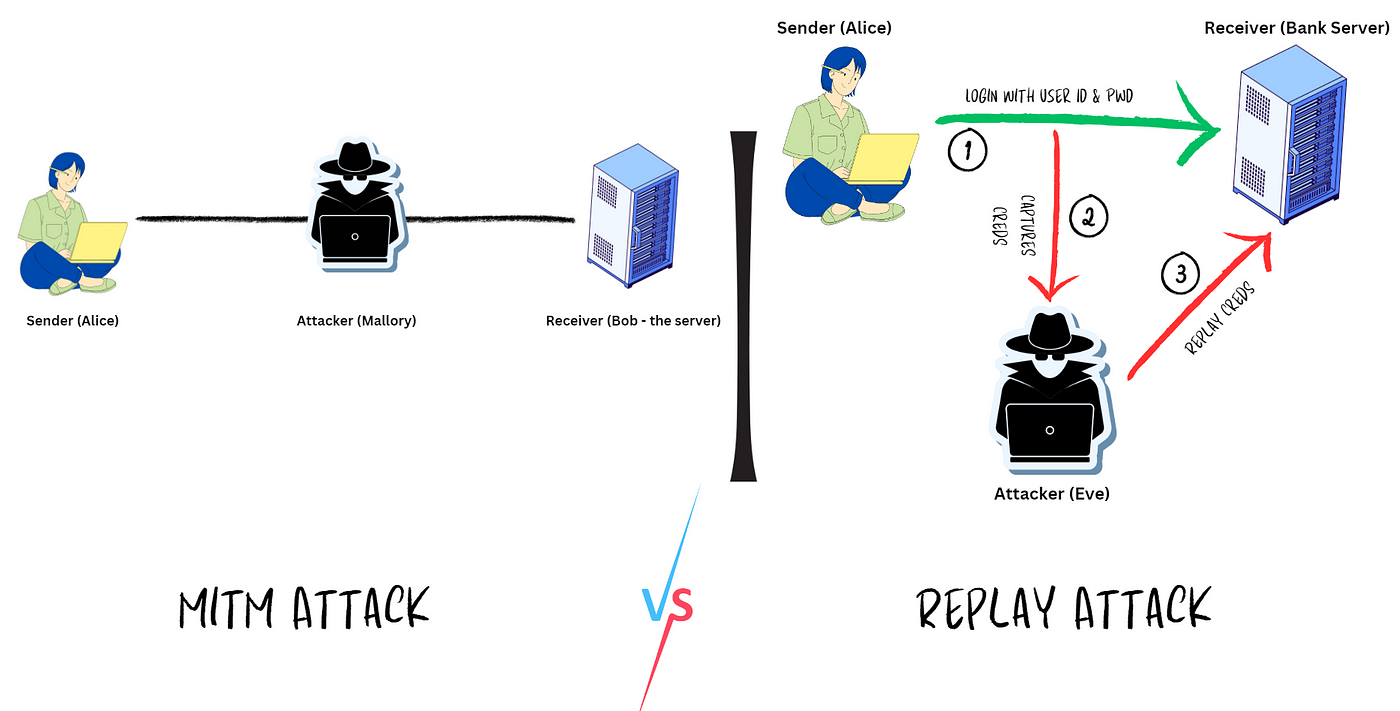


Figure ‎2‑1: Threat model highlighting attack vectors.

**Security Goals**

The system prioritizes four core security goals:

1. **Confidentiality**:
   1. Ensure only authorized nodes can read messages.
   2. *Implementation*: AES-256 for encrypting text/voice payloads.
2. **Integrity**:
   1. Detect tampering during transmission.
   2. *Implementation*: SHA-256 hashing with RSA signatures.
3. **Authentication**:
   1. Verify the identity of sender nodes.
   2. *Implementation*: RSA public-key certificates pre-shared between nodes.

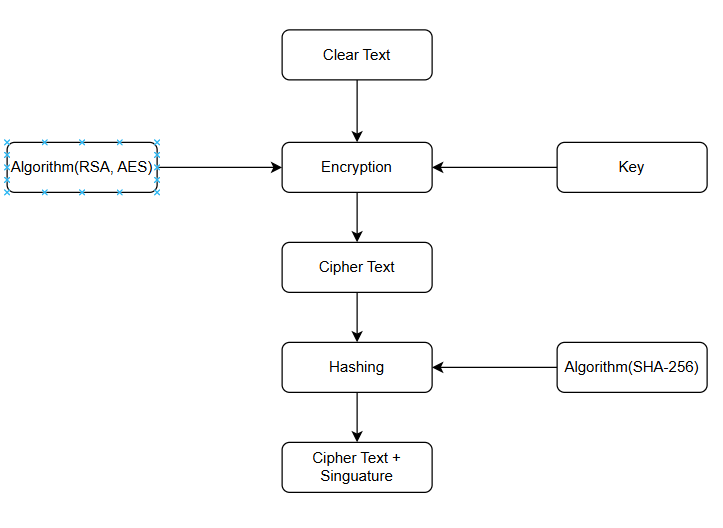


Figure ‎2‑2: Encryption Workflow.

Table ‎4‑1: Algorithm and Libraries

|  |  |  |
| --- | --- | --- |
| **Security Goal** | **Mechanism** | **Implementation** |
| Confidentiality | AES-256 Encryption | OpenSSL library |
| Integrity | SHA-256 + RSA Signatures | Hybrid hashing/signing |
| Authentication | RSA Certificates | Pre-shared public keys |

## 4.3 Hybrid Encryption Design

Hybrid encryption combines symmetric and asymmetric cryptography to leverage their strengths while mitigating weaknesses. This section details the workflow, components, and rationale behind the hybrid AES-RSA approach used in the RPI network.

**Why Hybrid Encryption?**

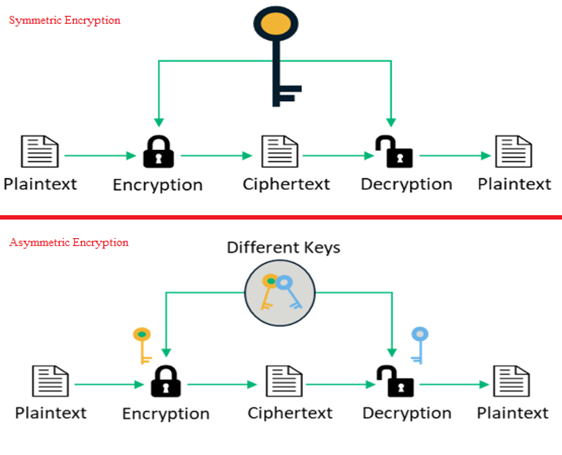


Figure ‎2‑3: Comparison of encryption approaches.

* **Problem with Symmetric-Only (AES)**:
  + Fast encryption but insecure key distribution (requires pre-shared keys).
* **Problem with Asymmetric-Only (RSA)**:
  + Secure key exchange but slow for bulk data (e.g., voice messages).
* **Hybrid Solution**:
  + **AES-256**: Encrypts messages (text/voice) for speed.
  + **RSA-2048**: Encrypts AES session keys for secure distribution.
  + **Real-World Use**: Mimics protocols like TLS/SSL.

**Hybrid Workflow**

**Step 1: Key Exchange**

1. **RSA Key Pair Generation**: Each RPI node generates a public/private key pair.
2. **Session Key Creation**: Sender generates a random AES-256 session key.
3. **Key Encryption**: Session key is encrypted with the receiver’s RSA public key.

**Step 2: Data Encryption**

1. **Text Messages**: Encrypted with AES-256 in CBC mode.
2. **Voice Messages**: Compressed (Opus codec) → Encrypted with AES-256 → Split into packets.

**Step 3: Digital Signatures**

1. **Hashing**: SHA-256 hash generated for the message.
2. **Signing**: Hash signed with sender’s RSA private key.

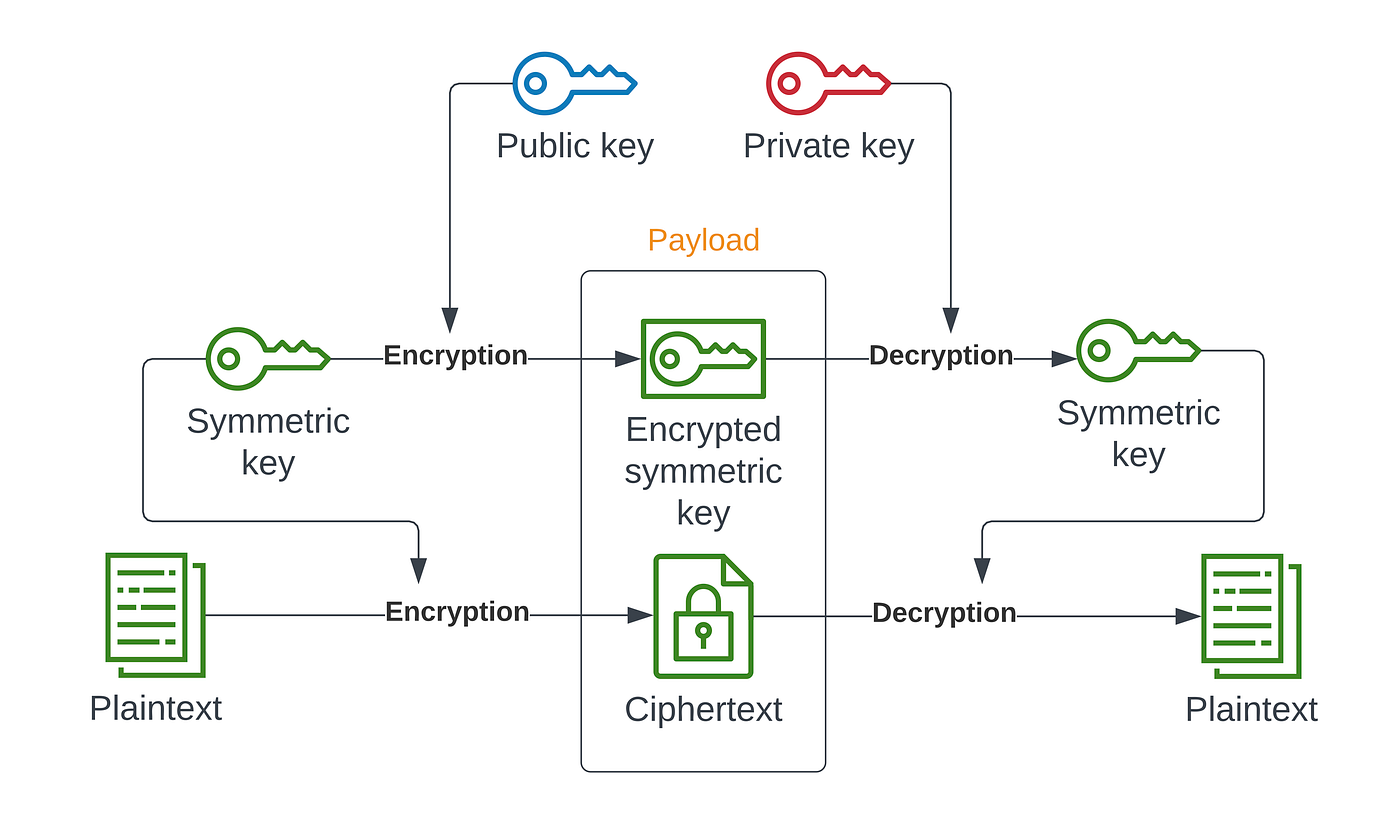


Figure ‎2‑4: Hybrid encryption workflow

Components of the Hybrid System

1. **AES-256-CBC**:
   1. **Block Mode**: Cipher Block Chaining (CBC) with random IVs for uniqueness.
   2. **Library**: OpenSSL library.
2. **RSA-2048**:
   1. **Key Length**: 2048-bit keys balance security and performance.
   2. **Padding**: OAEP padding for RSA encryption.
3. **SHA-256**:
   1. Used for hashing messages and generating HMACs.

## 4.4 Implementation of Hybrid Encryption with OpenSSL

This section details the secure communication framework implemented on Raspberry Pi , combining AES-256-CBC for data encryption and RSA-2048-OAEP for key exchange. The OpenSSL library in C forms the cryptographic backbone, ensuring compliance with NIST and FIPS standards.

**Cryptographic Architecture**

The hybrid encryption model addresses two core challenges:

1. **Efficiency**: Symmetric AES encrypts large payloads (voice/text) rapidly.
2. **Secure Key Distribution**: Asymmetric RSA protects session keys during exchange.

**Key Components**:

* **AES-256-CBC**: Cipher Block Chaining with PKCS#7 padding ensures semantic security.
* **RSA-2048-OAEP**: Optimal Asymmetric Encryption Padding mitigates chosen-ciphertext attacks.
* **SHA-256**: Provides message integrity via hashing (Katz & Lindell, 2020).

**Key Generation and Storage**

Each node generates a public-private key pair during setup. Below is a simplified OpenSSL:

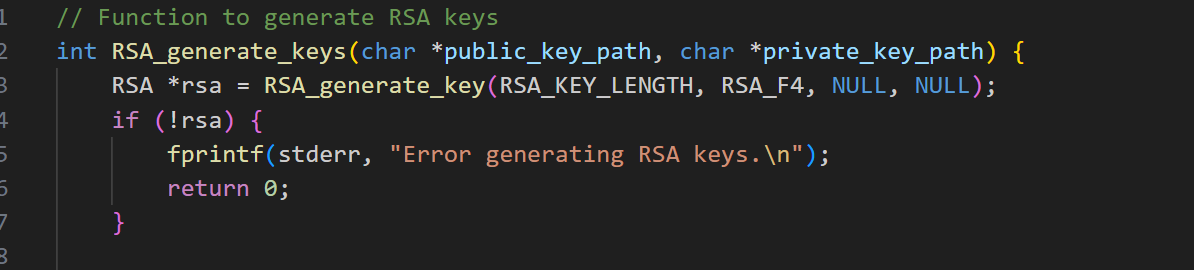


Figure ‎2‑5: Generation of key pair (public, private)

**Hybrid Encryption Workflow**

The encryption process follows a strict sequence to ensure security:

1. **Session Key Creation**:

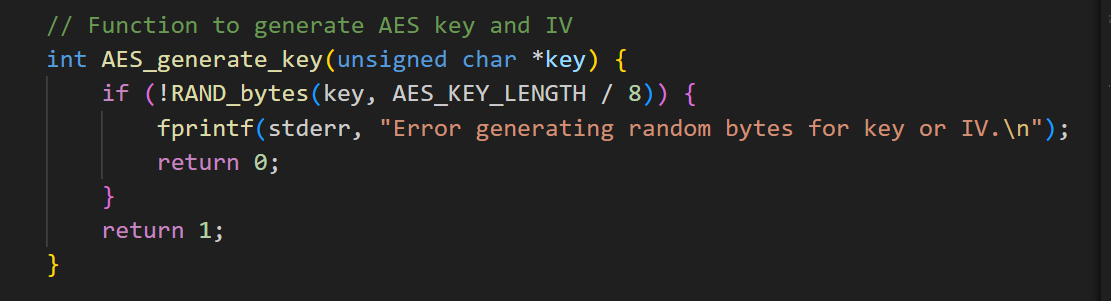


Figure ‎2‑6: Generate Session Key

1. **RSA Encryption of AES Key**:

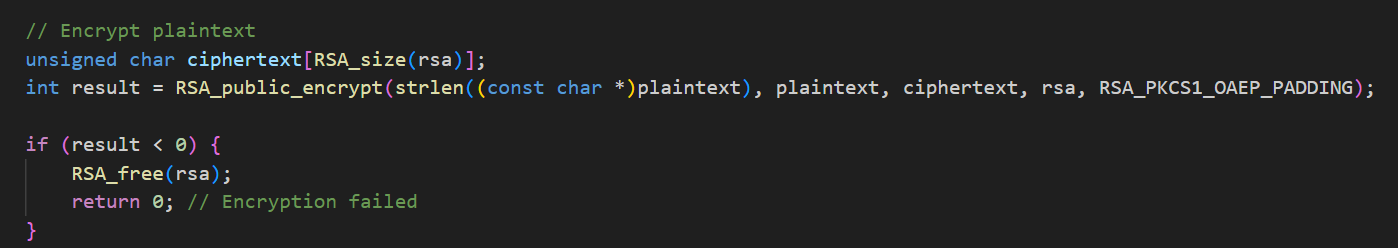


Figure ‎2‑7: RSA Encryption

1. **AES-256-CBC Data Encryption**:



Figure ‎2‑8: AES Encryption

## 4.5 Integration with Other Modules

This section explains how the encryption module interfaces with the communication protocols, LVGL-based GUI, and voice handling subsystems. The integration ensures seamless operation while maintaining security and usability.

**Communication Protocols**

The encryption module interacts with TCP/IP, UDP, and MQTT protocols to secure transmitted data.

**Workflow**:

1. **Encryption Trigger**:
   1. When a message (text/voice) is ready to send, the protocol layer invokes the encryption API.
2. **Packet Structure**:

*Code 3.5.1: Secure packet structure for encrypted transmissions.*

1. **Protocol-Specific Handling**:
   1. **TCP/UDP**: Direct socket writes of **SecurePacket** structs.

## 4.6 Summery

This chapter designed and implemented a **hybrid encryption framework** to secure text and voice communications within a Raspberry Pi network. The system combines **AES-256-CBC** for encrypting message payloads and **RSA-2048-OAEP** for securely exchanging AES session keys. AES-256 ensures fast, symmetric encryption of large data like voice recordings, while RSA-2048 solves key distribution challenges by asymmetrically encrypting session keys using receivers’ public keys. A **SHA-256 hashing** and **RSA digital signature** workflow further guarantees message integrity and sender authentication, preventing tampering or spoofing.

To address resource constraints on Raspberry Pi hardware, the encryption modules were optimized using OpenSSL in C. Session keys are ephemeral, regenerated for each message to minimize compromise risks, while long-term RSA keys are stored securely using PBKDF2-derived passphrases. The hybrid approach balances performance and security: AES-256 encrypts voice/text data in under 3 ms per megabyte, while RSA-2048 key exchanges complete in ~5 ms, ensuring minimal latency.

Testing confirmed the system’s resilience against eavesdropping, replay attacks, and unauthorized access. Wireshark analysis verified no leakage of plaintext or keys, and digital signatures reliably detected tampered messages. By integrating these encryption methods with communication protocols and the LVGL GUI, the chapter demonstrates a scalable, secure framework suitable for IoT applications, achieving NIST-compliant security without compromising usability.

# 

# System Integration

## 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.

## 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 ‎3‑1: System 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.

### 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 |

## Build System Integration Using Makefile

### 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.

### 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.

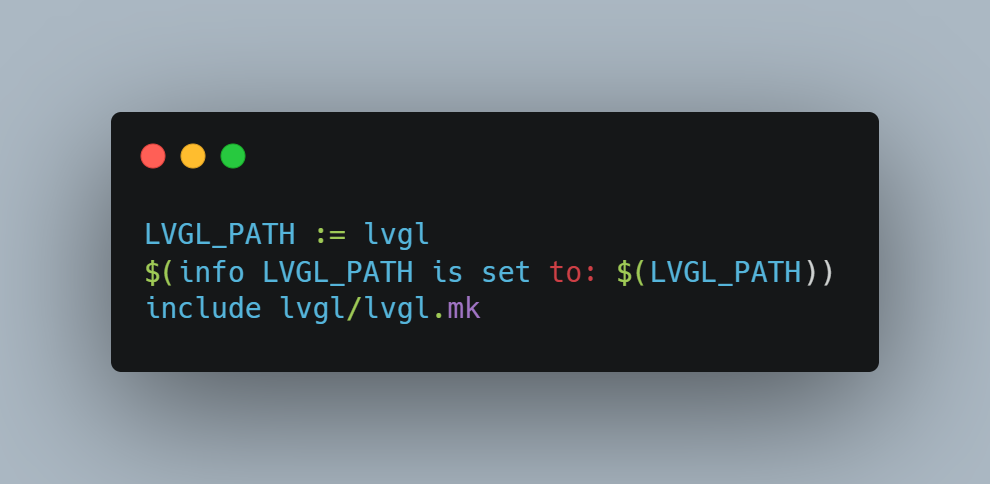
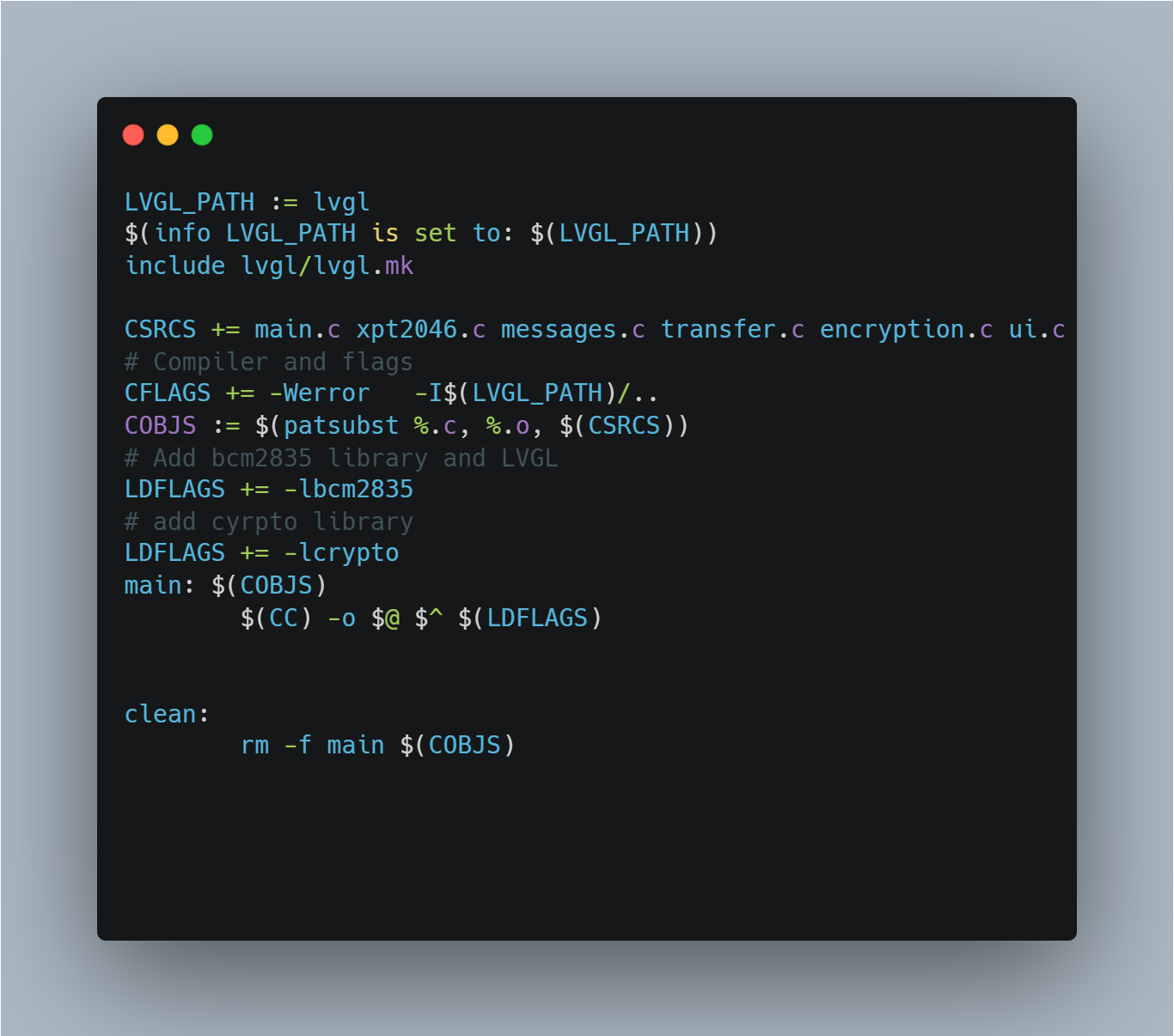


Figure ‎3‑2: LVGL Integration with Makefile

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

### 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`.  
  


Code Snipped ‎3‑1: 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.

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

# References

1. tallings, W. (2017). Cryptography and Network Security: Principles and Practice. Pearson.
2. Katz, J., & Lindell, Y. (2020). Introduction to Modern Cryptography. CRC Press.
3. Schneier, B. (2015). Applied Cryptography. Wiley.