****

**Kernel Module Development for Keypad Based Home Security System**

Submitted by,

T Tushar Shenoy 4NM21EC167

Tapan Nayak 4NM21EC169

U. Akash Shenoy 4NM21EC172

Shrijay Sanjay Bandekar 4NM21EC186

Under the guidance of

**Dr. SUKESH RAO**

**Associate Professor**

**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**N.M.A.M. INSTITUTE OF TECHNOLOGY, NITTE - 574110**

**2024 – 2025**

**Description**

The objective of this project is to develop a kernel module and an application for a Raspberry Pi to create a security system that allows users to set and reset a password via a keypad. The hardware setup includes a Raspberry Pi, a 4x4 keypad for password input, an LCD display that shows static messages, a buzzer for audio feedback, an LED indicator, and a servo motor for unlocking actions.

The kernel module is designed to manage the interactions between the keypad, the LCD, and other components. It utilizes IOCTL commands to securely set and reset the password, ensuring that it is stored and accessed only through the device file. The LCD displays two fixed messages: "Security System" and "Enter Password," providing basic guidance to the user during the password entry process.

Password submission is accomplished using the \* key on the keypad, while the # key is used to clear the current password entry. When a user enters a password, the system evaluates the input. If the password is correct, the buzzer emits two short beeps, the LED lights up for five seconds, and the servo motor rotates to indicate that the door is unlocking. Conversely, if the password is incorrect, the buzzer sounds five consecutive beeps to alert the user.

The application code provides a user interface for setting a new password or resetting the existing one. When the user inputs a password using the keypad, the application sends this information to the kernel module via a device file. However, the LCD does not update to reflect the status of the password input or validation; it remains static throughout the operation. This design simplifies the interaction with the system but limits feedback provided to the user during the password management process.

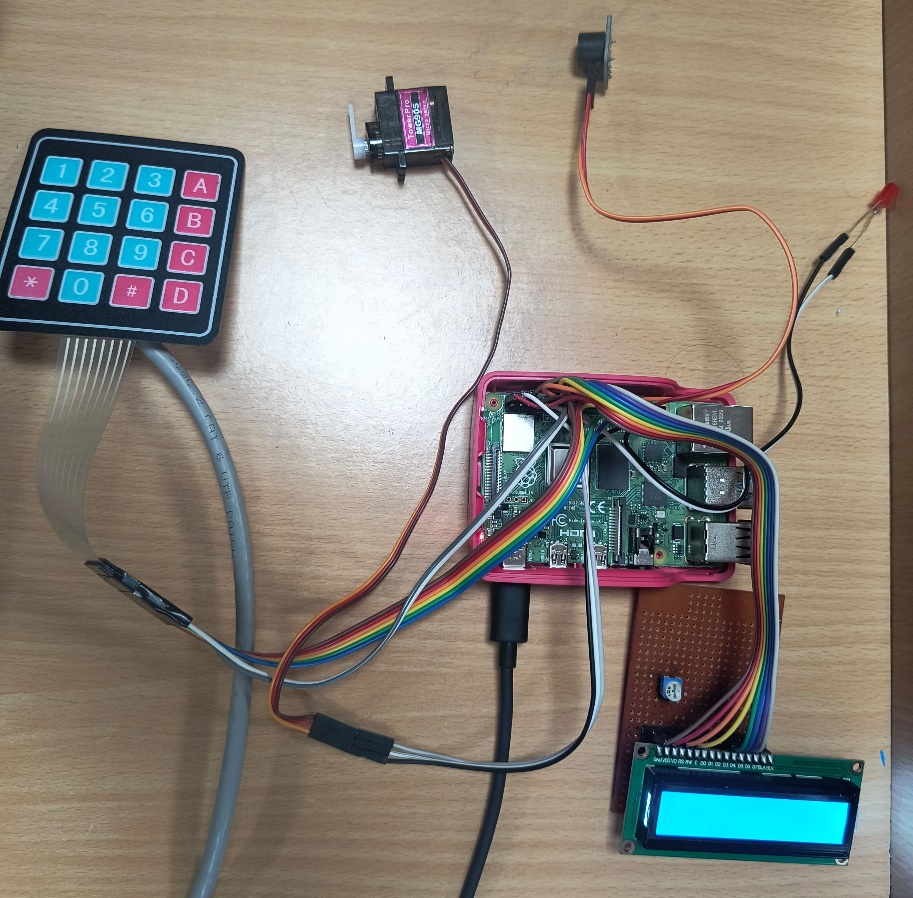


Fig. 1 Hardware setup

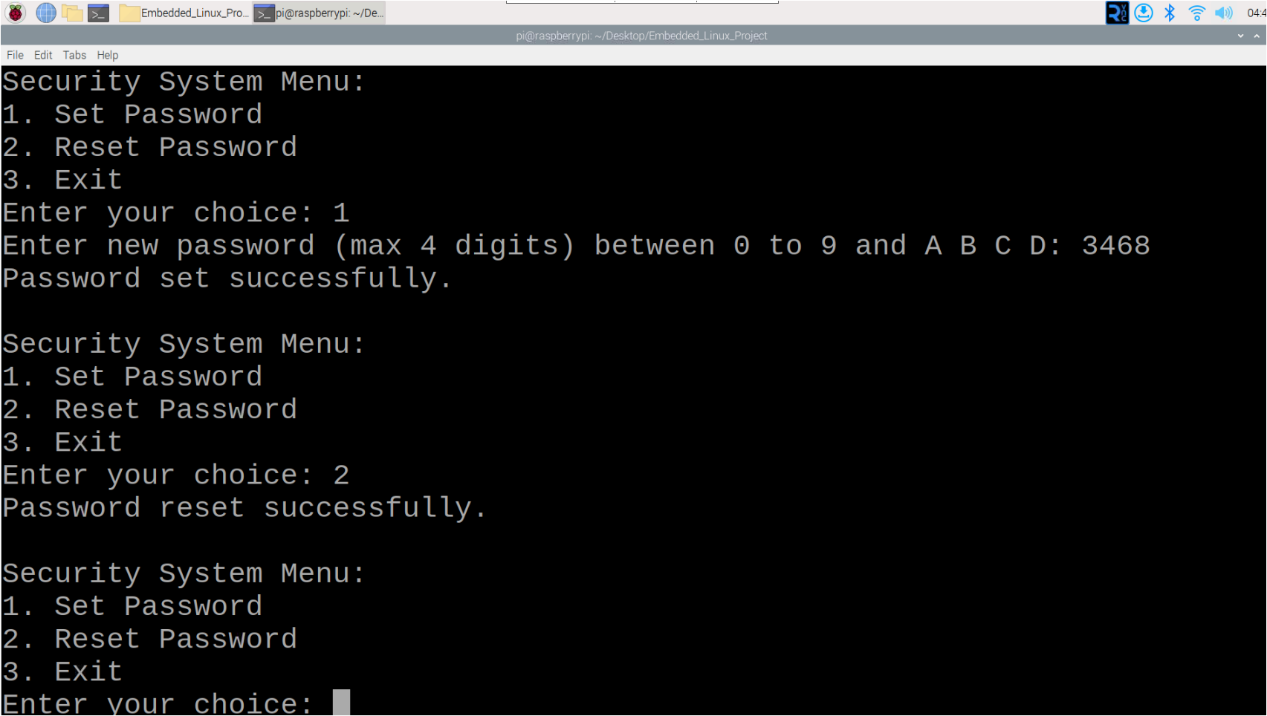


Fig.2 User Interface

**Code**

**securitysystem.c**

#include <linux/init.h> // For module initialization and cleanup macros

#include <linux/module.h> // For module macros such as MODULE\_LICENSE

#include <linux/gpio.h> // For GPIO access

#include <linux/delay.h> // For delay functions

#include <linux/string.h> // For string operations like strcmp

#include <linux/timer.h> // For kernel timer functionality

#include <linux/jiffies.h> // For using jiffies in the timer

#include <linux/fs.h> // For file operations and character device registration

#include <linux/uaccess.h> // For copying data from user space to kernel space

MODULE\_LICENSE("GPL"); // License type for the module

// Definitions for keypad and device settings

#define ROWS 4

#define COLS 4

#define PASSWORD "1234" // Default password

#define PASSWORD\_MAX\_LENGTH 5 // Maximum length for the password

#define LED\_PIN 17 // GPIO pin for the LED

#define BUZZER\_PIN 26 // GPIO pin for the buzzer

#define SERVO\_PIN 13 // GPIO pin for the servo motor

// LCD pin definitions

#define RS\_PIN 18

#define E\_PIN 23

#define D4\_PIN 24

#define D5\_PIN 25

#define D6\_PIN 8

#define D7\_PIN 7

// Servo PWM settings

#define SERVO\_MIN\_PULSE 500 // Minimum pulse width for the servo in microseconds

#define SERVO\_MAX\_PULSE 2400 // Maximum pulse width for the servo in microseconds

#define SERVO\_FREQ 50 // Frequency for the servo motor

// IOCTL command definitions

#define IOCTL\_SET\_PASSWORD \_IOW('a', 1, char \*)

#define IOCTL\_RESET\_PASSWORD \_IO('a', 2)

// Arrays defining row and column GPIO pins for the keypad

static unsigned int row\_pins[ROWS] = {4, 9, 10, 11};

static unsigned int col\_pins[COLS] = {5, 6, 27, 22};

// 4x4 Keypad mapping

static char keypad\_map[ROWS][COLS] = {

{'1', '2', '3', 'A'},

{'4', '5', '6', 'B'},

{'7', '8', '9', 'C'},

{'\*', '0', '#', 'D'}

};

// Variables to store the password and track entered keys

static char current\_password[PASSWORD\_MAX\_LENGTH] = PASSWORD; // Current password, default is "1234"

static struct timer\_list keypad\_timer; // Timer for keypad scanning

static int scan\_interval = 200; // Keypad scan interval in milliseconds

static char entered\_password[PASSWORD\_MAX\_LENGTH]; // Array to store the entered password

static int current\_index = 0; // Index to track the position in entered\_password

static int major\_number; // Major number for character device registration

// Function to send a command to the LCD

void lcd\_send\_command(char cmd);

// Function to send data (characters) to the LCD

void lcd\_send\_data(char data);

// Function to initialize the LCD

void lcd\_init(void);

// Function to print a string on the LCD

void lcd\_print(const char \*str);

// Function to make the buzzer beep a specified number of times

static void beep\_buzzer(int times, int duration) {

int i;

for (i = 0; i < times; i++) {

gpio\_set\_value(BUZZER\_PIN, 1); // Turn the buzzer on

mdelay(duration); // Wait for the specified duration

gpio\_set\_value(BUZZER\_PIN, 0); // Turn the buzzer off

if (i < times - 1) { // If more beeps are needed, wait between beeps

mdelay(duration);

}

}

}

// Function to generate PWM for the servo motor to set the angle

void servo\_pwm(int angle) {

int pulse\_width = SERVO\_MIN\_PULSE + (angle \* (SERVO\_MAX\_PULSE - SERVO\_MIN\_PULSE) / 180); // Calculate pulse width

gpio\_set\_value(SERVO\_PIN, 1); // Start the pulse

udelay(pulse\_width); // Keep the pulse on for the calculated width

gpio\_set\_value(SERVO\_PIN, 0); // End the pulse

udelay(20000 - pulse\_width); // Wait for the remainder of the 20ms period

}

// Timer callback function to scan the keypad

static void scan\_keypad(struct timer\_list \*t) {

int row, col;

for (row = 0; row < ROWS; row++) { // Iterate through each row

for (int r = 0; r < ROWS; r++) {

gpio\_set\_value(row\_pins[r], 1); // Set all row pins high

}

gpio\_set\_value(row\_pins[row], 0); // Set the current row low

for (col = 0; col < COLS; col++) { // Iterate through each column

if (!gpio\_get\_value(col\_pins[col])) { // If a key is pressed

char key\_pressed = keypad\_map[row][col]; // Get the key value from the keypad map

beep\_buzzer(1, 200); // Beep once to indicate key press

if (key\_pressed == '\*') { // '\*' key is used to submit the password

entered\_password[current\_index] = '\0'; // Null-terminate the entered password

if (strcmp(entered\_password, current\_password) == 0) { // Check if the entered password is correct

beep\_buzzer(2, 100); // Beep twice for correct password

gpio\_set\_value(LED\_PIN, 1); // Turn on the LED

servo\_pwm(130); // Rotate the servo to unlock position

mdelay(5000); // Wait for 5 seconds

servo\_pwm(30); // Rotate the servo back to lock position

gpio\_set\_value(LED\_PIN, 0); // Turn off the LED

} else { // Incorrect password

beep\_buzzer(5, 200); // Beep five times for incorrect password

}

current\_index = 0; // Reset the index for the next password

} else if (key\_pressed == '#') { // '#' key is used to reset the input

current\_index = 0; // Reset the index

memset(entered\_password, 0, sizeof(entered\_password)); // Clear the entered password

beep\_buzzer(1, 1000); // Long beep to indicate reset

} else if (current\_index < 4) { // If a valid key is pressed and there is space in the password buffer

entered\_password[current\_index++] = key\_pressed; // Add the key to the password buffer

}

}

}

}

mod\_timer(&keypad\_timer, jiffies + msecs\_to\_jiffies(scan\_interval)); // Re-arm the timer

}

// Function to initialize the LCD

void lcd\_init(void) {

gpio\_request(RS\_PIN, "RS\_PIN");

gpio\_request(E\_PIN, "E\_PIN");

gpio\_request(D4\_PIN, "D4\_PIN");

gpio\_request(D5\_PIN, "D5\_PIN");

gpio\_request(D6\_PIN, "D6\_PIN");

gpio\_request(D7\_PIN, "D7\_PIN");

gpio\_direction\_output(RS\_PIN, 0); // Set RS as output

gpio\_direction\_output(E\_PIN, 0); // Set E as output

gpio\_direction\_output(D4\_PIN, 0); // Set D4 as output

gpio\_direction\_output(D5\_PIN, 0); // Set D5 as output

gpio\_direction\_output(D6\_PIN, 0); // Set D6 as output

gpio\_direction\_output(D7\_PIN, 0); // Set D7 as output

mdelay(15); // Initial delay for LCD

lcd\_send\_command(0x33); // Initialize LCD in 4-bit mode

lcd\_send\_command(0x32); // Set to 4-bit mode

lcd\_send\_command(0x28); // 2 line, 5x7 matrix

lcd\_send\_command(0x0C); // Display ON, cursor OFF

lcd\_send\_command(0x01); // Clear display

mdelay(2); // Wait for the command to execute

}

}

// Function to send a command to the LCD

void lcd\_send\_command(char cmd) {

gpio\_set\_value(RS\_PIN, 0); // Command mode

// Send higher nibble

gpio\_set\_value(D4\_PIN, (cmd >> 4) & 1);

gpio\_set\_value(D5\_PIN, (cmd >> 5) & 1);

gpio\_set\_value(D6\_PIN, (cmd >> 6) & 1);

gpio\_set\_value(D7\_PIN, (cmd >> 7) & 1);

gpio\_set\_value(E\_PIN, 1); // Enable pulse

mdelay(1);

gpio\_set\_value(E\_PIN, 0); // Disable pulse

mdelay(1);

// Send lower nibble

gpio\_set\_value(D4\_PIN, cmd & 1);

gpio\_set\_value(D5\_PIN, (cmd >> 1) & 1);

gpio\_set\_value(D6\_PIN, (cmd >> 2) & 1);

gpio\_set\_value(D7\_PIN, (cmd >> 3) & 1);

gpio\_set\_value(E\_PIN, 1); // Enable pulse

mdelay(1);

gpio\_set\_value(E\_PIN, 0); // Disable pulse

mdelay(1);

}

// Function to print a string

void lcd\_print(const char \*str) {

while (\*str) {

gpio\_set\_value(RS\_PIN, 1);

gpio\_set\_value(D4\_PIN, (\*str >> 4) & 1);

gpio\_set\_value(D5\_PIN, (\*str >> 5) & 1);

gpio\_set\_value(D6\_PIN, (\*str >> 6) & 1);

gpio\_set\_value(D7\_PIN, (\*str >> 7) & 1);

gpio\_set\_value(E\_PIN, 1);

mdelay(1);

gpio\_set\_value(E\_PIN, 0);

mdelay(1);

gpio\_set\_value(D4\_PIN, \*str & 1);

gpio\_set\_value(D5\_PIN, (\*str >> 1) & 1);

gpio\_set\_value(D6\_PIN, (\*str >> 2) & 1);

gpio\_set\_value(D7\_PIN, (\*str >> 3) & 1);

gpio\_set\_value(E\_PIN, 1);

mdelay(1);

gpio\_set\_value(E\_PIN, 0);

mdelay(1);

str++;

}

}

// IOCTL function to handle commands for setting and resetting the password

static long securitysystem\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

char new\_password[PASSWORD\_MAX\_LENGTH];

switch (cmd) {

case IOCTL\_SET\_PASSWORD:

if (copy\_from\_user(new\_password, (char \_\_user \*)arg, PASSWORD\_MAX\_LENGTH)) {

return -EFAULT;

}

strncpy(current\_password, new\_password, PASSWORD\_MAX\_LENGTH);

printk(KERN\_INFO "Password updated successfully.\n");

break;

case IOCTL\_RESET\_PASSWORD:

strncpy(current\_password, PASSWORD, PASSWORD\_MAX\_LENGTH);

printk(KERN\_INFO "Password reset to default.\n");

break;

default:

return -EINVAL;

}

return 0;

}

// File operations structure for the character device

static struct file\_operations fops = {

.unlocked\_ioctl = securitysystem\_ioctl,

};

// Module initialization function

static int securitysystem\_init(void) {

int ret, i;

major\_number = register\_chrdev(0, "securitysystem", &fops);

if (major\_number < 0) {

printk(KERN\_ALERT "Failed to register character device\n");

return major\_number;

}

printk(KERN\_INFO "Registered character device with major number %d\n", major\_number);

lcd\_init();

lcd\_print("Security system");

lcd\_send\_command(0xC0);

lcd\_print("Enter password");

ret = gpio\_request(LED\_PIN, "LED\_PIN");

if (ret) {

printk(KERN\_ALERT "Failed to request GPIO %d for LED\n", LED\_PIN);

return ret;

}

gpio\_direction\_output(LED\_PIN, 0);

ret = gpio\_request(BUZZER\_PIN, "BUZZER\_PIN");

if (ret) {

printk(KERN\_ALERT "Failed to request GPIO %d for Buzzer\n", BUZZER\_PIN);

return ret;

}

gpio\_direction\_output(BUZZER\_PIN, 0);

ret = gpio\_request(SERVO\_PIN, "SERVO\_PIN");

if (ret) {

printk(KERN\_ALERT "Failed to request GPIO %d for servo\n", SERVO\_PIN);

return ret;

}

gpio\_direction\_output(SERVO\_PIN, 0);

for (i = 0; i < ROWS; i++) {

ret = gpio\_request(row\_pins[i], "Row Pin");

if (ret) {

printk(KERN\_ALERT "Failed to request GPIO %d for row pin\n", row\_pins[i]);

return ret;

}

gpio\_direction\_output(row\_pins[i], 1);

}

for (i = 0; i < COLS; i++) {

ret = gpio\_request(col\_pins[i], "Col Pin");

if (ret) {

printk(KERN\_ALERT "Failed to request GPIO %d for column pin\n", col\_pins[i]);

return ret;

}

gpio\_direction\_input(col\_pins[i]);

}

timer\_setup(&keypad\_timer, scan\_keypad, 0);

mod\_timer(&keypad\_timer, jiffies + msecs\_to\_jiffies(scan\_interval));

return 0;

}

// Module cleanup function

static void securitysystem\_exit(void) {

int i;

del\_timer(&keypad\_timer);

unregister\_chrdev(major\_number, "securitysystem");

gpio\_free(LED\_PIN);

gpio\_free(BUZZER\_PIN);

gpio\_free(SERVO\_PIN);

for (i = 0; i < ROWS; i++) {

gpio\_free(row\_pins[i]);

}

for (i = 0; i < COLS; i++) {

gpio\_free(col\_pins[i]);

}

lcd\_send\_command(0x01);

}

module\_init(securitysystem\_init);

module\_exit(securitysystem\_exit);

**app.c**

#include <stdio.h> // Standard I/O functions

#include <stdlib.h> // Standard library functions

#include <string.h> // String manipulation functions

#include <fcntl.h> // File control functions

#include <unistd.h> // POSIX API functions

#include <sys/ioctl.h> // Device control operations

#define DEVICE\_PATH "/dev/securitysystem" // Device file path

#define PASSWORD\_MAX\_LENGTH 5 // Max password length

// Custom IOCTL commands

#define IOCTL\_SET\_PASSWORD \_IOW('a', 1, char \*) // Set password command

#define IOCTL\_RESET\_PASSWORD \_IO('a', 2) // Reset password command

// Display menu options

void display\_menu() {

printf("\nSecurity System Menu:\n"); // Menu header

printf("1. Set Password\n"); // Set password option

printf("2. Reset Password\n"); // Reset password option

printf("3. Exit\n"); // Exit option

}

int main() {

int choice, fd; // User choice and file descriptor

char password[PASSWORD\_MAX\_LENGTH]; // Password storage

fd = open(DEVICE\_PATH, O\_RDWR); // Open device file

if (fd < 0) { // Check for errors

perror("Failed to open device"); // Error message

return EXIT\_FAILURE; // Exit on failure

}

while (1) { // Infinite loop

display\_menu(); // Show menu

printf("Enter your choice: "); // Prompt choice

scanf("%d", &choice); // Read choice

switch (choice) { // Handle choices

case 1: // Set password

printf("Enter new password (max 4 digits) between 0 to 9 and A B C D: "); // Prompt input

scanf("%4s", password); // Read password

password[PASSWORD\_MAX\_LENGTH - 1] = '\0'; // Null-terminate

if (ioctl(fd, IOCTL\_SET\_PASSWORD, password) < 0) { // Set password via ioctl

perror("Failed to set password"); // Error if ioctl fails

} else {

printf("Password set successfully.\n"); // Success message

}

break; // Exit switch

case 2: // Reset password

if (ioctl(fd, IOCTL\_RESET\_PASSWORD) < 0) { // Reset via ioctl

perror("Failed to reset password"); // Error if ioctl fails

} else {

printf("Password reset successfully.\n"); // Success message

}

break; // Exit switch

case 3: // Exit program

close(fd); // Close device file

printf("Exiting...\n"); // Exit message

return EXIT\_SUCCESS; // Exit successfully

default: // Invalid choice

printf("Invalid choice. Please try again.\n"); // Inform user

break; // Exit switch

}

}

}

**Makefile**

obj-m := securitysystem.o

all:

make -C/lib/modules/6.1.21-v8+/build M=$(PWD) modules

clean:

make -C/lib/modules/6.1.21-v8+/build M=$(PWD) clean

**setup.sh**

# Set GPIO pins 5, 6, 27, and 22 to pull-up mode

raspi-gpio set 5 pu # Enable pull-up on GPIO pin 5

raspi-gpio set 6 pu # Enable pull-up on GPIO pin 6

raspi-gpio set 27 pu # Enable pull-up on GPIO pin 27

raspi-gpio set 22 pu # Enable pull-up on GPIO pin 22

# Clean previous build files

make clean # Remove any existing compiled files

# Compile the kernel module and application

make all # Compile all components

# Insert the security system kernel module

sudo insmod securitysystem.ko # Load the kernel module into the kernel

# Compile the application code

gcc -o app app.c # Compile the app.c file into an executable named ‘app’

# Create a device file for the security system

sudo mknod /dev/securitysystem c 236 0 # Create a character device file with major number 236 and minor number 0

# Run the application

sudo ./app # Execute the compiled application

**Steps to Use**

1. **Create a Project Folder**
   * Create a new folder where you will place all the project files.
2. **Create and Add Code to Files**
   * Inside the folder, create a file named securitysystem.c. This will contain the kernel module code. Paste the securitysystem.c code into this file.
   * Next, create a file named app.c. This will contain the application code. Paste the app.c code into this file.
   * Similarly, create two more files:
     + Makefile: Add the Makefile contents to this file for compiling your code.
     + setup.sh: Add the setup script contents to this file to automate the setup process.
3. **Give Access to the Script File**
   * Before running setup.sh, provide execution permissions using:

sudo chmod 777 setup.sh

1. **Run the Script File**
   * Execute the setup script to compile and insert the kernel module:

sudo ./setup.sh

* + This will compile the kernel module, load it and start its operation.

1. **Using the Application**
   * The application will run, and you can interact with it. To exit from the kernel module operation, type 3 when prompted in the application and press Enter.

This will successfully set up and run your security system project on the Raspberry Pi.