Calculator

Pothuri Rahul - EE24BTECH11050

1 Introduction

This project presents a calculator using Arduino , embedded C language . This calculator is designed so that we can compute many mathematical operations like arithmetic operations , trigonometric functions , inverse trigonometric functions , power function , etc.This should be done without using math.h library , hemce used some different methods to compute few functions.

2 Hardware things

2.1 Components used :

- Breadboard
- LCD display
- Arduino board
- Connecting cables and wires
- Mobile
- push buttons

2.2 Circuit design

2.2.1 LCD to Arduino

Following connections are done between LCD and Arduino

LCD display	Arduino
RS	8
Enable(E)	9
D_4	10
D_5	11
D_6	12
D_7	13
RW	GND
VSS	GND
VDD	5 V
VEE	Potentiometer (Middle Pin)

2.2.2 Push buttons to Arduino

Let us consider each 6 buttons as a row and their are 6 such rows (A total of 36 buttons) . Each row and column are connected as follows .

ROW	**Arduino**
Row 1	2
Row 2	3
Row 3	4
Row 4	5
Row 5	6
Row 6	7

Column	**Arduino**
Column 1	A_0
Column 2	A_1
Column 3	A_2
Column 4	A_3
Column 5	A_4
Column 6	A_5

Functionality of each push button is as follows.

Row	**Col 1 (A0)**	**Col 2 (A1)**	**Col 3 (A2)**	**Col 4 (A3)**	**Col 5 (A4)**	**Col 6 (A5)**
R1 (2)	0	6	*	sin	e	$\mathrm{y^x}$
R2 (3)	1	7	/	cos	log	z
R3 (4)	2	8	=	tan	ln	π
R4 (5)	3	9	C (Clear)	cot	sqrt	natural log
R5 (6)	4	+	i	sec	cosec	Answer
R6 (7)	5	-	.	sin^{-}	cos^{-}	tan^{-}

Table 1: 6×6 Keypad Button Mapping

3 Software things

3.1 Code used

The c code used is the following

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

// TYPEDEFS
typedef uint8_t byte;

// LCD PINS
#define ClearBit(x,y) x &= ~_BV(y)
#define SetBit(x,y) x |= _BV(y)
#define LCD_RS 0
#define LCD_E 1
#define DAT4 2
#define DAT5 3
#define DAT6 4
#define DAT7 5
#define CLEARDISPLAY 0x01

#define ROWS 6
#define COLS 6

#define K 0.607252935
#define NUM_STEPS 15

// Define the macro before using it
#define ANGLE_OUT_DEGREES { \
    45.00000000000000,   /* atan(2^(-0))  */ \
    26.56505117707799,   /* atan(2^(-1))  */ \
    14.03624346792648,   /* atan(2^(-2))  */ \
    7.125016348901798,   /* atan(2^(-3))  */ \
    3.576334374997351,   /* atan(2^(-4))  */ \
    1.789910608246069,   /* atan(2^(-5))  */ \
    0.8951737102110744,  /* atan(2^(-6))  */ \
    0.4476141708605531,  /* atan(2^(-7))  */ \
    0.2238105003685381,  /* atan(2^(-8))  */ \
    0.1119056770662069,  /* atan(2^(-9))  */ \
    0.05595289189380368, /* atan(2^(-10)) */ \
    0.02797645261700364, /* atan(2^(-11)) */ \
    0.01398822714226501 6,/* atan(2^(-12)) */ \
    0.006994113675352919,/* atan(2^(-13)) */ \
    0.003497056850704011  /* atan(2^(-14)) */ \
}

// Then use it to initialize the array
double angle_out_degrees[] = ANGLE_OUT_DEGREES;

void PulseEnableLine(void) {
    SetBit(PORTB, LCD_E);
    _delay_us(40);
    ClearBit(PORTB, LCD_E);
}

void SendNibble(byte data) {
    PORTB &= 0xC3; // Clear 4 data lines
```

```
    if (data & _BV(4)) SetBit(PORTB, DAT4);
    if (data & _BV(5)) SetBit(PORTB, DAT5);
    if (data & _BV(6)) SetBit(PORTB, DAT6);
    if (data & _BV(7)) SetBit(PORTB, DAT7);
    PulseEnableLine();
}

void SendByte(byte data) {
    SendNibble(data);
    SendNibble(data << 4);
}

void LCD_Cmd(byte cmd) {
    ClearBit(PORTB, LCD_RS);
    SendByte(cmd);
}

void LCD_Char(byte ch) {
    SetBit(PORTB, LCD_RS);
    SendByte(ch);
}

void LCD_Init() {
    LCD_Cmd(0x33);
    LCD_Cmd(0x32);
    LCD_Cmd(0x28);
    LCD_Cmd(0x0C);
    LCD_Cmd(0x06);
    LCD_Cmd(0x01);
    _delay_ms(3);
}

void LCD_Clear() {
    LCD_Cmd(CLEARDISPLAY);
    _delay_ms(3);
}

void LCD_Message(const char *text) {
    while (*text) LCD_Char(*text++);
}

void LCD_Float(double data) {
    char st[16];
    dtostrf(data, 6, 2, st);
    LCD_Message(st);
}

// VARIABLES
volatile double Num1 = 0.0, Num2 = 0.0, Number = 0.0;
volatile double answer = 0.0; // Added variable to store the last result
volatile char action = 0;
volatile uint8_t result = 0;
volatile uint8_t inputMode = 0;
volatile bool opt1 = 0, opt2 = 0, decimal_mode = false, shift = false;
volatile double decimal_factor = 0.1;
volatile uint8_t return_act = 0;
volatile bool math_error = false;  // New flag to track math errors
double x = 0;
double y = 0;
```

```
double angle = 0.0;  // Added missing global variable declaration

// BUTTONS MAPPING
uint8_t row_pins[ROWS] = {PD0, PD1, PD2, PD3, PD4, PD5};
uint8_t col_pins[COLS] = {PC0, PC1, PC2, PC3, PC4, PC5};

char keypad[ROWS][COLS] = {
    {'0', '6', '*', 's', 'e', 'y'},
    {'1', '7', '/', 'c', 'l', 'z'},
    {'2', '8', '=', 't', 'a', 'p'},
    {'3', '9', 'C', 'q', 'b', 'L'},
    {'4', '+', 'i', 'm', 'w', 'A'},
    {'5', '-', '.', 'n', 'x', 'S'}
};

void keypad_init() {
    // Configure row pins as input with pull-up resistors (Now using PORTD)
    for (int i = 0; i < ROWS; i++) {
        DDRD &= ~(1 << row_pins[i]); // Set as input
        PORTD |= (1 << row_pins[i]); // Enable pull-up
    }

    // Configure column pins as output
    for (int j = 0; j < COLS; j++) {
        DDRC |= (1 << col_pins[j]);  // Set as output
        PORTC |= (1 << col_pins[j]); // Set high
    }
}

uint8_t keypad_scan() {
    for (int col = 0; col < COLS; col++) {
        // Set all columns high
        for (int j = 0; j < COLS; j++) {
            PORTC |= (1 << col_pins[j]);
        }

        // Set current column low
        PORTC &= ~(1 << col_pins[col]);

        for (int row = 0; row < ROWS; row++) {
            if (!(PIND & (1 << row_pins[row]))) { // Key is pressed
                _delay_ms(50); // Debounce
                while (!(PIND & (1 << row_pins[row]))); // Wait for release
                return keypad[row][col]; // Return key value
            }
        }
    }
    return 0; // No key pressed
}

double ln(double x) {
    if (x <= 0) {
        math_error = true; // Set math error flag
        return 0; // Logarithm undefined for non-positive numbers
    }

    double y = x - 1.0; // Initial approximation
    for (int i = 0; i < 1000; i++) {  // More iterations improve accuracy
        y = y - (exp(y) - x) / exp(y) - 1;
```

```
    }
    return y;
}

double exp(double x) {
    double y = 1.0; // Initial condition: e^0 = 1
    double h = 0.01; // Step size for approximation
    int steps = (int)(x / h); // Number of steps

    if (x < 0) { // Handle negative exponent using inverse
        x = -x;
        for (int i = 0; i < steps; i++) {
            y *= (1 + h);
        }
        return 1.0 / y;
    }

    for (int i = 0; i < steps; i++) {
        y *= (1 + h);
    }

    return y;
}

// Custom square root function since we can't use math.h
double my_sqrt(double x) {
    if (x < 0) {
        math_error = true;
        return 0;
    }

    if (x == 0) return 0;

    // Newton's method for square root
    double guess = x / 2.0;
    double prev_guess;

    for (int i = 0; i < 10; i++) {
        prev_guess = guess;
        guess = (guess + x / guess) / 2.0;

        // Check for convergence
        if (fabs(guess - prev_guess) < 0.0001)
            break;
    }

    return guess;
}

// Improved power function
double power(double base, double exponent) {
    // Special case: anything^0 = 1 (except 0^0 which is undefined)
    if (exponent == 0) {
        if (base == 0) {
            math_error = true; // 0^0 is undefined
            return 0;
        }
        return 1.0;
    }
```

```

    // Special case: 0^anything = 0 (except 0^negative which is undefined)
    if (base == 0) {
        if (exponent < 0) {
            math_error = true; // Division by zero
            return 0;
        }
        return 0;
    }

    // Handle integer exponents directly for better accuracy
    bool is_int_exponent = (fabs(exponent - round(exponent)) < 0.000001);
    double int_exp = round(exponent);

    if (is_int_exponent && int_exp > 0 && int_exp <= 10) {
        // Direct calculation for small positive integer exponents
        double result = 1.0;
        for (int i = 0; i < int_exp; i++) {
            result *= base;
        }
        return result;
    }

    // Handle negative base with integer exponent
    if (base < 0) {
        if (!is_int_exponent) {
            math_error = true; // Complex result for non-integer exponent
            return 0;
        }

        // For negative base with integer exponent
        double result = exp(exponent * ln(-base));
        return (fmod(int_exp, 2) == 0) ? result : -result;
    }

    // Use logarithm method for other cases
    return exp(exponent * ln(base));
}

void sin_cos(double n) {
    // Declare necessary variables
    x = 1.0;
    y = 0.0;
    double angle = 0.0;

    // CORDIC algorithm for sin and cos calculation
    for (uint8_t i = 0; i < NUM_STEPS; i++) {
        int sigma = (angle < n) ? 1 : -1;

        double scale = 1.0 / (1UL << i);  // Precompute scale = 2^-i
        double x_new = x - sigma * (y * scale);
        double y_new = y + sigma * (x * scale);

        x = x_new;
        y = y_new;

        angle += sigma * angle_out_degrees[i];
    }

```

```
    x = x * K;
    y = y * K;
}

void inv_trigo(double z, char mode) {
    // Declare necessary variables
    x = 1.0;
    y = 0.0;
    angle = 0.0;

    // Input validation: z must be between -1 and 1 for arcsin/arccos
    if ((mode == 'a' || mode == 'b') && (z < -1.0 || z > 1.0)) {
        math_error = true; // Set math error flag
        return;
    }

    // Initialize starting point based on the inverse function type
    switch (mode) {
        case 'a': // arcsin
            x = my_sqrt(1-z*z);
            y = z;
            break;
        case 'b': // arccos
            x = z;
            y = my_sqrt(1-z*z);
            break;
        case 'w': // arctan
            x = 1;
            y = z;
            break;
        case 'x': // arccot
            x = z;
            y = 1;
            break;
        case 'y': // arccsc
            if (fabs(z) < 1.0) {
                math_error = true; // Set math error flag
                return; // Invalid domain
            }
            x = my_sqrt(z*z-1)/fabs(z);
            y = 1/fabs(z);
            if (z < 0) y = -y;
            break;
        case 'z': // arcsec
            if (fabs(z) < 1.0) {
                math_error = true; // Set math error flag
                return; // Invalid domain
            }
            x = 1/fabs(z);
            y = my_sqrt(z*z-1)/fabs(z);
            if (z < 0) x = -x;
            break;
            // Handle unexpected values
    }

    // Now rotate back to the x-axis
    for (uint8_t i = 0; i < NUM_STEPS; i++) {
        // Determine rotation direction to reduce y toward 0
        int sigma = (y < 0) ? 1 : -1;
```

```c

        double scale = 1.0 / (1UL << i);  // Precompute scale = 2^-i
        double x_new = x - sigma * (y * scale);
        double y_new = y + sigma * (x * scale);

        x = x_new;
        y = y_new;

        // Accumulate rotation angle
        angle += sigma * angle_out_degrees[i];
    }
}

void calculate_result() {
    math_error = false; // Reset error flag at start of calculation

    if (action == '+') Number = Num1 + Num2;
    else if (action == '-') Number = Num1 - Num2;
    else if (action == '*') Number = Num1 * Num2;
    else if (action == 'p') Number = power(Num1, Num2);
    else if (action == 'h') {
        if (Num2 == 0) {
            math_error = true;
            Number = 0;
        } else if (Num2 == 2) {
            Number = my_sqrt(Num1); // Square root (2nd root)
        } else {
            // For nth root, use: x^(1/n)
            Number = power(Num1, 1.0/Num2);
        }
    }
    else if (action == 'L') {
        if (Num1 <= 0 || Num2 <= 0 || fabs(Num1 - 1.0) < 0.000001) {
            math_error = true; // Set math error flag for invalid inputs
            Number = 0;
        } else {
            Number = ln(Num2)/ln(Num1);
        }
    }
    else if (action == '/') {
        if (Num2 == 0) {
            math_error = true; // Set math error flag
            Number = 0;
        } else {
            Number = Num1 / Num2;
        }
    }
    else if (action == 's' || action == 'c' || action == 't' || action == 'q' || action == 'm' || action == 'n')
        sin_cos(Num2);
        switch(action) {
            case 's': Number = y;  // sine
                    break;
            case 'c': Number = x;  // cosine
                    break;
            case 't':
                    if (fabs(x) < 0.000001) { // Check for division by zero (cos  0)
                        math_error = true;
                        Number = 0;
                    } else {
```

```
                        Number = y/x;  // tangent
                    }
                    break;
            case 'q':
                    if (fabs(y) < 0.000001) { // Check for division by zero (sin  0)
                        math_error = true;
                        Number = 0;
                    } else {
                        Number = x/y;  // cotangent
                    }
                    break;
            case 'm':
                    if (fabs(y) < 0.000001) { // Check for division by zero (sin  0)
                        math_error = true;
                        Number = 0;
                    } else {
                        Number = 1/y;  // cosecant
                    }
                    break;
            case 'n':
                    if (fabs(x) < 0.000001) { // Check for division by zero (cos  0)
                        math_error = true;
                        Number = 0;
                    } else {
                        Number = 1/x;  // secant
                    }
                    break;
        }
    }
    else if (action == 'a' || action == 'b' || action == 'w' || action == 'x' || action == 'y' || action == 'z')
        inv_trigo(Num2, action);
        Number = -angle;
    }
    else if (action == 'r' ) {
        Number = (exp(Num2) - exp(-Num2))/2;
    }
    else if (action == 'g' ) {
        Number = (exp(Num2) + exp(-Num2))/2;
    }
    else if(action == 'e'){
        Number = exp(Num2);
    }
    else if(action == 'l'){
        Number = ln(Num2);
    }

    // Store the result in the answer variable if no math error occurred
    if (!math_error) {
        answer = Number;
    }

    result = 1;
}

void process_input(char key) {
    if (key == 'i') {
        shift = !shift;
    }
    else if (key == 'S') {
```

```
        // Toggle the sign of the current number
        Number = -Number;
    }
    else if (key == 'A') {
        // Recall the stored answer value
        Number = answer;
        LCD_Clear();
        LCD_Message("Ans: ");
        LCD_Float(answer);
        _delay_ms(1000);
    }
    else if (key == 'C') {
    if(shift == 0){
        Number = Num1 = Num2 = 0;
        action = 0;
        result = 0;
        decimal_mode = false;
        decimal_factor = 0.1;
        shift = 0;
        math_error = false; // Reset error flag
        // Note: We don't reset the answer variable to preserve it across calculations
        LCD_Clear();
        LCD_Message("Cleared");
        _delay_ms(1000);
        LCD_Clear();
        return;
        }
        else{
         if (!decimal_mode) {
        // Remove last digit in integer mode
        Number = (Number < 0) ? -((int)(-Number / 10)) : (int)(Number / 10);
    } else {
        // Separate integer and decimal parts
        double intPart, fracPart;
        fracPart = modf(Number, &intPart);

        // Track decimal precision
        static int decimal_places = 0;

        if (fracPart != 0) {
            // If we have decimal digits, remove the last one
            decimal_places--;

            if (decimal_places <= 0) {
                // If no more decimal places, switch back to integer mode
                Number = intPart;
                decimal_mode = false;
                decimal_places = 0;
            } else {
                // Round to the remaining decimal places
                double multiplier = power(10, decimal_places);
                fracPart = round(fracPart * multiplier) / multiplier;
                Number = (Number < 0) ? -(abs(intPart) + abs(fracPart)) : (intPart + fracPart);
            }
        } else {
            // No decimal part, switch back to integer mode
            decimal_mode = false;
            decimal_places = 0;
        }
```

```
    }
        }
    }
    else if (key >= '0' && key <= '9') {
        if (result) {  // If a calculation was done, start fresh
            Number = 0;
            result = 0;
            decimal_mode = false;
            decimal_factor = 0.1;
            math_error = false; // Reset error flag
        }
        if (!decimal_mode) {
            Number = (Number * 10) + (key - '0');
            // Preserve the sign when adding digits
            if (Number < 0) {
                Number = -fabs(Number);
            }
        } else {
            // Add decimal digits while preserving sign
            if (Number < 0) {
                Number -= (key - '0') * decimal_factor;
            } else {
                Number += (key - '0') * decimal_factor;
            }
            decimal_factor *= 0.1;
        }
    }
    else if (key == '.') {
        if (!decimal_mode) {
            decimal_mode = true;
        }
    }

    else if (key == '+' || key == '-' || key == '*' || key == '/'  || key == 'L') {
        Num1 = Number;
        Number = 0;
        action = key;
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false; // Reset error flag
    }

else if (key == 'p'){
    if(shift == 0){
        action = 'p';
        Num1 = Number;
        Number = 0;
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false;
    }
    else{
        action = 'h'; // Root operation
        Num1 = Number; // Store the number we want to find the root of
        Number = 0;    // Reset number for entering the root order
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false;
        // Don't reset shift here - let the display show we're in root mode
```

```
    }
}

else if (key == 'p'){
    if(shift == 0){
        action = 'p';
        Num1 = Number;
        Number = 0;
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false;
    }
    else{
        action = 'h'; // Root operation
        Num1 = Number; // Store the number we want to find the root of
        Number = 0;    // Reset number for entering the root order
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false;
        // Don't reset shift here - let the display show we're in root mode
    }
}

    else if (key == 's' || key == 'c' || key == 't' || key == 'q' || key == 'm' || key == 'n') {
        if(shift == 0){
            switch(key) {
                case 's': action = 's';
                        break;
                case 'c': action = 'c';
                        break;
                case 't': action = 't';
                        break;
                case 'q': action = 'q';
                        break;
                case 'm': action = 'm';
                        break;
                case 'n': action = 'n';
                        break;
            }
        }
        else{
            switch(key) {
                case 's': action = 'r';
                        break;
                case 'c': action = 'g';
                        break;
            }
        }
    }
    else if(key == 'a' || key == 'b' || key == 'w' || key == 'x' || key == 'y' || key == 'z'){
        switch(key) {
            case 'a': action = 'a';
                    break;
            case 'b': action = 'b';
                    break;
            case 'w': action = 'w';
                    break;
            case 'x': action = 'x';
                    break;
```

```
            case 'y': action = 'y';
                    break;
            case 'z': action = 'z';
                    break;
        }

        Number = 0;  // Reset Number to avoid displaying 0.00
        decimal_mode = false;
        decimal_factor = 0.1;
        math_error = false; // Reset error flag
    }
    else if(key == 'e'){
        if(shift == 0){
            action = 'e';
            Number = 0;  // Reset Number to avoid displaying 0.00
            decimal_mode = false;
            decimal_factor = 0.1;
            math_error = false; // Reset error flag
        }
        else {
            Number = exp(1);  // Set Number to e
            shift = 0;        // Reset shift flag after using it

            // Special flag to indicate we should display 'e' instead of the value
            action = 'E';     // Using capital 'E' to indicate special display mode
        }
    }
    else if(key == 'l'){
        if(shift == 0){
            action = 'l';
            Number = 0;  // Reset Number to avoid displaying 0.00
            decimal_mode = false;
            decimal_factor = 0.1;
            math_error = false; // Reset error flag
        }
        else {
            Number = 3.14159265358979;  // Set Number to pi
            shift = 0;        // Reset shift flag after using it

            // Special flag to indicate we should display '' instead of the value
            action = 'P';     // Using capital 'L' to indicate special display mode for pi
        }
    }
    else if (key == '=') {
        Num2 = Number;
        calculate_result();
        decimal_mode = false;
        decimal_factor = 0.1;
    }
}

int main(void) {
    keypad_init();
    DDRB |= (1 << LCD_RS) | (1 << LCD_E) | (1 << DAT4) | (1 << DAT5) | (1 << DAT6) | (1 << DAT7);
    LCD_Init();
    LCD_Clear();
    LCD_Float(Number);

    while (1) {
```

```
        char key = keypad_scan();
        if (key) {
            process_input(key);
            LCD_Clear();

            if (result == 1) {
                result = 0;

                if (math_error) {
                    // Display "Math Error" if a math error occurred
                    LCD_Message("Math Error");
                    _delay_ms(1500); // Display error message a bit longer

                    // Reset all calculation values except answer
                    Number = Num1 = Num2 = 0;
                    action = 0;
                    decimal_mode = false;
                    decimal_factor = 0.1;
                    math_error = false;

                    LCD_Clear();
                    LCD_Float(0); // Display zero after error
                } else {
                    // No error, display result normally
                    LCD_Message("Result:");
                    _delay_ms(100);
                    LCD_Float(Number);
                    _delay_ms(100);
                }
            }
            else if (action) {
                // Fixed display logic with proper structure
                if (action == 'E') {
                    LCD_Message("e");
                }
                else if (action == 'h') {  // Root display
        if (Num2 != 0) {
            // When Num2 is set, display the complete expression
            LCD_Float(Num2);
            LCD_Message("-root(");
            LCD_Float(Num1);
            LCD_Message(")");
        } else {
            // When just starting the root operation
            LCD_Message("root(");
            LCD_Float(Num1);
            LCD_Message(",");
            if (Number != 0) {
                LCD_Float(Number);
            }
        }
    }
                else if (action == 'p') {
                    LCD_Float(Num1);
                    LCD_Message("^");
                    if (Number != 0) {
                        LCD_Float(Number);
                    }
                }
```

```
                else if (action == 'L' && shift == 0) {  // Regular logarithm
                    LCD_Message("log_");
                    LCD_Float(Num1);
                    LCD_Message("(");
                    if (Number != 0) {
                        LCD_Float(Number);
                        LCD_Message(")");
                    }
                }
                else if (action == 'P') {  // Pi constant
                    LCD_Message("pi");
                }
                // First check for inverse trig functions (they need special handling)
                else if (action == 'e') {
                    LCD_Message("exp(");
                    LCD_Float(Number);
                    LCD_Message(")");
                }
                else if (action == 'l') {
                    LCD_Message("ln(");
                    LCD_Float(Number);
                    LCD_Message(")");
                }
                else if (action == 'a' || action == 'b' || action == 'w' || action == 'x' || action == 'y' || act
                    switch(action) {
                        case 'a': LCD_Message("asin("); break;
                        case 'b': LCD_Message("acos("); break;
                        case 'w': LCD_Message("atan("); break;
                        case 'x': LCD_Message("acot("); break;
                        case 'y': LCD_Message("acosec("); break;
                        case 'z': LCD_Message("asec("); break;
                    }
                    if (Number != 0) {
                        LCD_Float(Number);
                        LCD_Message(")");
                    }
                }
                // Then check for standard trig functions
                else if (action == 's' || action == 'c' || action == 't' || action == 'q' || action == 'm' || act
                    switch(action) {
                        case 's': LCD_Message("sin("); break;
                        case 'c': LCD_Message("cos("); break;
                        case 't': LCD_Message("tan("); break;
                        case 'q': LCD_Message("cot("); break;
                        case 'm': LCD_Message("cosec("); break;
                        case 'n': LCD_Message("sec("); break;
                        case 'r': LCD_Message("sinh("); break;
                        case 'g': LCD_Message("cosh("); break;
                    }
                    if (Number != 0) {
                        LCD_Float(Number);
                        LCD_Message(")");
                    }
                }
                // Finally handle basic operations
                else {
                    LCD_Float(Num1);
                    _delay_ms(100);
                    LCD_Char(action);
```

```
                    _delay_ms(100);
                    if (Number != 0) {
                        LCD_Float(Number);
                    }
                }
            } else {
                LCD_Float(Number);
                _delay_ms(100);
            }
        }
    }
}
```

Reference : Used code done by Yamasani Harsha Vardhan Reddy - ee24btech11063.

3.2 Explanation of the code

The given Arduino code implements a digital clock (HH:MM:SS) using a 7447 BCD to 7-segment decoder with multiplexing.

3.2.1 Key Features

1. **LCD Handling (4-bit Mode)**
    - Initializes and controls an **LCD display** using **PORTB**.
    - Functions like `LCD_Init()`, `LCD_Clear()`, and `LCD_Message()` handle output.
2. **Keypad Scanning (6×6 Matrix)**
    - `keypad_init()`: Configures **6 row inputs (PORTD)** and **6 column outputs (PORTC)**.
    - `keypad_scan()`: Detects pressed buttons by scanning rows and columns.
3. **Mathematical Operations**
    - Implements basic (`+`, `-`, `*`, `/`) and advanced operations (`log`, `exp`, `sqrt`).
    - **Trigonometry** (`sin`, `cos`, `tan`) and **inverse functions** (`asin`, `acos`) are computed using the **CORDIC algorithm**.
4. **Handling User Input**
    - `process_input(char key)`: Determines the action based on the pressed button.
    - Supports **Shift mode** for accessing extra functions ($\sin^{-1}$, e^x, π).
    - Pressing `=` triggers `calculate_result()`, which processes the stored operation.

5. **Result Calculation & Display**
   - `calculate_result()`: Executes the selected mathematical function.
   - Displays results on **LCD**, with **error handling** for invalid inputs (e.g., division by zero).
6. **Memory & Error Handling**
   - Stores the **previous answer** (`ANS`).
   - Handles **math errors** like log(0) or division by zero.

3.2.2 How It Works

1. **User presses buttons** → Detected via `keypad_scan()`.
2. **Input is processed** → Determines the type of operation.
3. **Calculation is performed** → `calculate_result()` executes math operations.
4. **Result is displayed** → LCD shows the computed value.
5. **Shift Mode** allows access to extra functions (e.g., $\sin^{-1}$, `log`).

4 Future Enhancements

- Adding differentiation and integration operations .
- Finding zeros of given expression or solution of a given equation .
- Adding charging battery system to make the calculator portable.
- Adding degree or radian mode i.e, by using a toggle button between them .

5 Conclusion

By doing this project I learned how to use c language to built mathematical functions without using math.h . This project successfully implemented a fully functional calculator using an Arduino Uno, push buttons , and LCD display.

Acknowledgment

I thank Yamasani Harsha Vardhan Reddy - ee24btech11063 for helping me in code.