Scientific Calculator using AVR-GCC through Arduino

EE1003: Scientific Programming for Electrical Engineers

Y. Harsha Vardhan Reddy EE24BTECH11063

Contents

1 Introduction

The scientific calculator implemented in this project is built using an AVR microcontroller, programmed with C using the AVR-GCC compiler. The calculator provides a wide range of functions including:

- Basic arithmetic operations (addition, subtraction, multiplication, division)
- Trigonometric functions (sine, cosine, tangent, cotangent, secant, cosecant)
- Inverse trigonometric functions
- Hyperbolic functions (sinh, cosh)
- Exponential and logarithmic functions
- Powers and roots
- Constants (π, e)
- Result storage and recall

The calculator employs custom implementations of mathematical functions rather than relying on standard libraries, making it more suitable for resource-constrained microcontrollers.

2 Hardware Components

2.1 Required Components

- Arduino
- LCD display
- Push buttons
- Connecting wires

2.2 Pin Connections

2.2.1 LCD Connections

The LCD is connected to PORTB of the microcontroller:

- LCD_RS (Register Select) $\rightarrow$ PORTB0
- LCD_E (Enable) $\rightarrow$ PORTB1
- Data line 4 $\rightarrow$ PORTB2
- Data line 5 $\rightarrow$ PORTB3

- Data line 6 → PORTB4
- Data line 7 → PORTB5

The LCD is operated in 4-bit mode to save microcontroller pins.

2.2.2 Keypad Connections

The 6×6 matrix keypad requires 12 pins for interfacing:

- Row pins (6) → PORTD0 to PORTD5
- Column pins (6) → PORTC0 to PORTC5

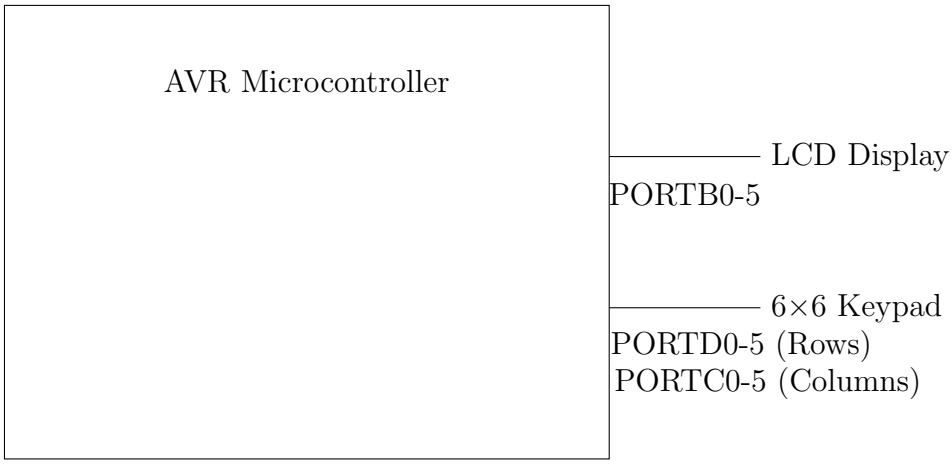

Figure 1: Hardware Connection Diagram

2.3 Keypad Layout

The calculator uses a 6×6 matrix keypad with the following key mapping:

0	1	2	3	4	5
6	7	8	9	+	-
*	/	=	Clear	shift	.
sin /**sinh**	cos /**cosh**	tan	cot	csc	sec
exp /**e**	ln /**pi**	arcsin	arccos	arctan	arccot
arccsc	arcsec	pow/**root**	log	Ans	Sign

- The functions in **bold** letters are computed when shift button is enabled

3 Software Architecture

3.1 Code Structure

The software is structured into several functional modules:

- LCD interface functions
- Keypad scanning functions
- Mathematical function implementations
- Input processing and calculation logic
- Main program loop

3.2 LCD Interface

The LCD interface uses a 4-bit mode to communicate with the HD44780 compatible display. The following functions are implemented:

- `PulseEnableLine()`: Generates the enable pulse for the LCD
- `SendNibble()`: Sends 4 bits of data to the LCD
- `SendByte()`: Sends a full byte by calling SendNibble twice
- `LCD_Cmd()`: Sends a command to the LCD
- `LCD_Char()`: Sends a character to the LCD
- `LCD_Init()`: Initializes the LCD in 4-bit mode
- `LCD_Clear()`: Clears the LCD display
- `LCD_Message()`: Displays a string on the LCD
- `LCD_Float()`: Displays a floating point number on the LCD

3.3 Keypad Scanning

The keypad scanning mechanism uses a matrix scanning approach:

- `keypad_init()`: Initializes the row pins as inputs with pull-ups and column pins as outputs
- `keypad_scan()`: Scans the keypad by activating one column at a time and checking row pins for key presses

The function returns the character associated with the pressed key according to the defined keypad mapping.

3.4 Mathematical Function Implementations

3.4.1 CORDIC Algorithm for Trigonometric and Inverse-Trigonometric Functions

The trigonometric and inverse-trigonometric functions are implemented which is explained in the following section 6

3.4.2 Exponential Function using Forward-Euler method

The forward Euler method is a first-order numerical approach to approximate solutions to ordinary differential equations. For the exponential function, we start with the differential equation:

$$\frac{dy}{dx} = y, \quad y(0) = 1 \tag{1}$$

This differential equation describes $y = e^x$. The Forward Euler method approximates this by:

$$y_{n+1} = y_n + h \cdot f(x_n, y_n) = y_n + h \cdot y_n = y_n(1 + h) \tag{2}$$

Where h is the step size. If we take n steps of size $h = \frac{x}{n}$ from $x = 0$ to x, we get:

$$y_n = y_0 \cdot \left(1 + \frac{x}{n}\right)^n \tag{3}$$

Logarithmic Function using Newton-Raphson Newton-Raphson is an iterative method for finding roots of equations. For calculating $\ln(x)$, we need to find y such that:

$$e^y - x = 0 \tag{4}$$

ref The Newton-Raphson iteration formula is:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} \tag{5}$$

Here, $f(y) = e^y - x$ and $f'(y) = e^y$. Substituting:

$$y_{n+1} = y_n - \frac{e^{y_n} - x}{e^{y_n}} \tag{6}$$

This is precisely the formula you provided. Starting with a good initial guess y_0 (often $y_0 = x - 1$ for $x > 0$ or a simpler approximation like $y_0 = \sqrt{x}$), this method converges quadratically to $\ln(x)$. The formula can be rewritten as:

$$y_{n+1} = y_n - 1 + \frac{x}{e^{y_n}} \tag{7}$$

3.4.3 Power and Root Functions

Power and root calculations use the exponential and logarithmic functions:

$$x^y = e^{y \cdot \ln(x)} \tag{8}$$

For roots:

$$\sqrt[n]{x} = x^{1/n} \tag{9}$$

3.5 Input Processing

The *process_input*() function handles key presses and maintains the calculator state:

- Number entry and decimal point handling
- Operation selection
- Calculation execution
- Result display
- Error handling

3.6 Main Program Loop

The main program continuously scans the keypad, processes key presses, and updates the display accordingly.

4 Mathematical Capabilities

4.1 Basic Arithmetic Operations

- Addition: `Number = Num1 + Num2`
- Subtraction: `Number = Num1 - Num2`
- Multiplication: `Number = Num1 * Num2`
- Division: `Number = Num1 / Num2`

4.2 Trigonometric Functions

- Sine: `sin_cos(Num2); Number = y;`
- Cosine: `sin_cos(Num2); Number = x;`
- Tangent: `sin_cos(Num2); Number = y/x;`
- Cotangent: `sin_cos(Num2); Number = x/y;`
- Secant: `sin_cos(Num2); Number = 1/x;`
- Cosecant: `sin_cos(Num2); Number = 1/y;`

4.3 Inverse Trigonometric Functions

- Arcsine: `inv_trigo(Num2, 'a'); Number = -angle;`
- Arccosine: `inv_trigo(Num2, 'b'); Number = -angle;`
- Arctangent: `inv_trigo(Num2, 'w'); Number = -angle;`
- Arccotangent: `inv_trigo(Num2, 'x'); Number = -angle;`
- Arcsecant: `inv_trigo(Num2, 'z'); Number = -angle;`
- Arccosecant: `inv_trigo(Num2, 'y'); Number = -angle;`

4.4 Hyperbolic Functions

- Hyperbolic sine: `Number = (exp(Num2) - exp(-Num2))/2;`
- Hyperbolic cosine: `Number = (exp(Num2) + exp(-Num2))/2;`

4.5 Exponential and Logarithmic Functions

- Exponential function: `Number = exp(Num2);`
- Natural logarithm: `Number = ln(Num2);`
- Logarithm with custom base: `Number = ln(Num2)/ln(Num1);`

4.6 Power and Root Functions

- Power: `Number = power(Num1, Num2);`
- nth Root: `Number = power(Num1, 1.0/Num2);`

4.7 Constants

- π (pi): 3.14159265358979
- e (Euler's number): exp(1)

5 Special Features

5.1 Error Handling

The calculator implements comprehensive error checking for:

- Division by zero
- Logarithm of non-positive numbers

- Square root of negative numbers
- Invalid domains for trigonometric functions
- Other mathematical errors

When errors occur, the calculator displays "Math Error" and resets the calculation state.

5.2 Result Storage

The calculator stores the last valid calculation result in the `answer` variable, which can be recalled using the "Ans" key.

5.3 Sign Toggle

The sign of the current number can be toggled using the dedicated "Sign" key.

5.4 Shift Function

The calculator implements a shift function to access secondary operations on certain keys:

- `power` → `root` when shifted
- `exp` → `e constant` when shifted
- `ln` → `pi constant` when shifted
- `sin` → `sinh` when shifted
- `cos` → `cosh` when shifted

6 CORDIC Algorithm Explained

This implementation uses the CORDIC (COordinate Rotation DIgital Computer) algorithm to calculate various trigonometric functions. CORDIC is particularly useful for systems with limited computational resources as it avoids complex multiplications and divisions.

6.1 Basic CORDIC Principle

CORDIC works by performing a series of rotations using only shifts and additions to compute trigonometric functions. The code shows two main functions:

1. `sin_cos()` - Computes sine and cosine
2. `inv_trigo()` - Computes inverse trigonometric functions

6.2 Trigonometric Functions Implementation

6.2.1 Basic equation of CORDIC's algorithm

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos\alpha \begin{bmatrix} 1 & \tan\alpha \\ -\tan\alpha & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- For making the computations economical(involving division by '2'), we took angles such that $\tan\alpha$ such that they are powers of $\frac{1}{2}$ and pre-computed the value of product of cosines accordingly.

6.2.2 Vector Rotation for Sine and Cosine

The `sin_cos()` function calculates sine and cosine simultaneously through a series of micro-rotations:

```
void sin_cos(double n) {
    // Initialize vector [1,0] and angle
    x = 1.0;
    y = 0.0;
    double angle = 0.0;

    // Perform CORDIC iterations
    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;
}
```

Key aspects:

- Starts with vector $[1, 0]$ and rotates it to reach target angle n
- Uses conditional rotation: clockwise if current angle $<$ target, counterclockwise otherwise

- Scale factor $1/(2^i)$ corresponds to $\arctan(2^{-i})$ angles and finally angle tends to n **This step makes this method very suitable for hardware since it is just dividing by 2 in each iteration which is just shifting of bits, which is very economical**
- K is a scaling constant which is multiplication of $cos(arctan(1/2^i))$
- After iterations, x contains cosine and y contains sine of angle n

6.2.3 Inverse Trigonometric Functions

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

    // Input validation
    if ((mode == 'a' || mode == 'b') && (z < -1.0 || z > 1.0)) {
        math_error = true;
        return;
    }

    // Set initial vector based on function type
    switch (mode) {
        case 'a': // arcsin
            x = my_sqrt(1-z*z);
            y = z;
            break;
        // ... other cases for arccos, arctan, etc.
    }

    // Rotate vector back to x-axis
    for (uint8_t i = 0; i < NUM_STEPS; i++) {
        int sigma = (y < 0) ? 1 : -1;

        double scale = 1.0 / (1UL << 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];
    }
}
```

Key aspects:

- Different initialization based on the inverse function type (arcsin, arccos, etc.)
- Domain validation for functions with restricted input ranges
- Rather than rotating to a target angle, it rotates to reduce y-component to zero
- Final angle accumulator contains the resulting inverse trigonometric value

6.3 Implementation Details

6.3.1 Rotation Logic

The core of CORDIC lies in its rotation logic:

1. **Sign Selection (σ)**: Determines rotation direction
   - For direct trig: `sigma = (angle < n) ?  1 :  -1`
   - For inverse trig: `sigma = (y < 0) ?  1 :  -1`
2. **Microrotation Step**:

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

   This implements rotation by approximately $\tan^{-1}(2^{-i})$ degrees
3. **Angle Tracking**:

   ```
   angle += sigma * angle_out_degrees[i];
   ```

   Accumulates rotation angles using a lookup table `angle_out_degrees`

6.3.2 Function Selection and Initialization

The `inv_trigo()` function cleverly handles six inverse trigonometric functions through different initializations:

- **arcsin**: Maps $z \to [x = \sqrt{1 - z^2}, y = z]$
- **arccos**: Maps $z \to [x = z, y = \sqrt{1 - z^2}]$
- **arctan**: Maps $z \to [x = 1, y = z]$
- **arccot**: Maps $z \to [x = z, y = 1]$

- **arccsc**: Maps $z \to$ specialized transformation with domain check
- **arcsec**: Maps $z \to$ specialized transformation with domain check

Each initialization creates a vector that, when rotated to the x-axis, produces the desired inverse function result as the accumulated angle.

7 Code Optimizations

7.1 Memory Usage Optimization

The code optimizes memory usage by:

- Using 4-bit mode for LCD interface (saves 4 pins)
- Reusing variables where possible
- Using appropriate data types (uint8_t for small integers)

7.2 Performance Optimization

Several performance optimizations are implemented:

- Pre-computed values for trigonometric calculations
- Early termination in iterative algorithms when sufficient precision is reached
- Special case handling for common inputs
- Efficient key debouncing

8 User Interface

8.1 Display Formatting

The calculator provides informative display output:

- Operation indicator showing current function
- Error messages for invalid operations
- Floating point display with adjustable precision
- Special symbols for mathematical constants (pi, e)

8.2 Input Handling

The calculator provides a responsive interface with:

- Decimal point input handling
- Negative number support
- Shift functionality for accessing secondary functions
- Clear function to reset calculations

9 Conclusion

The scientific calculator implementation demonstrates how to create a complex mathematical device using limited microcontroller resources. By employing efficient algorithms like CORDIC and custom numerical approximations, the calculator provides a wide range of mathematical functions without relying on floating-point libraries. The modular code structure separates hardware interfacing, mathematical algorithms, and user interface handling, making the code maintainable and extensible. The comprehensive error checking ensures robust operation, preventing issues with invalid operations. This implementation can serve as a starting point for more advanced calculators or as an educational tool for understanding algorithm implementation on microcontrollers.