# Scientific Calculator

EE24BTECH11049
Patnam Shariq Faraz Muhammed

**Abstract**

This report presents a comprehensive analysis of our Arduino-based scientific calculator implementation. The calculator utilizes the Shunting Yard algorithm for efficient expression parsing and employs RK4 for custom mathematical function calculations. We explore the electronic circuit design and software architecture, emphasizing the expression evaluation process, optimized mathematical computations, and an intuitive user interface.

CONTENTS

# 1 Introduction

The calculator project implements a functional and extendable scientific calculator capable of evaluating complex mathematical expressions with proper operator precedence. Key features include:

The calculator utilizes the Shunting Yard algorithm for expression evaluation and implements mathematical functions using RK4 method and numerical approximations. By avoiding external libraries, it ensures precise calculations while maintaining a minimal memory footprint, making it well-suited for the constrained AVR microcontroller environment.

# 2 Hardware Components

The following are components required for the project

- Arduino UNO
- Breadboard
- 36 push buttons
- Potentiometer
- Jumper wires
- Cell phone (to power the Arduino)

# 3 Connections

| **Signal/Pin Name** | **Arduino Connection** |
|---|---|
| **LCD Display** | |
| LCD_E (Enable) | PB1 |
| DB4 (Data Bit 4) | Pin 2 |
| DB5 (Data Bit 5) | Pin 3 |
| DB6 (Data Bit 6) | Pin 4 |
| DB7 (Data Bit 7) | Pin 5 |
| **Push Button** | |
| ROW signals | PORTC (DDR: DDRC, PIN: PINC) |
| COLUMN signals | PORTD (DDR: DDRD, PIN: PIND) |
| Number of buttons | 36 |

TABLE 0: Arduino and LCD Push Button Connections

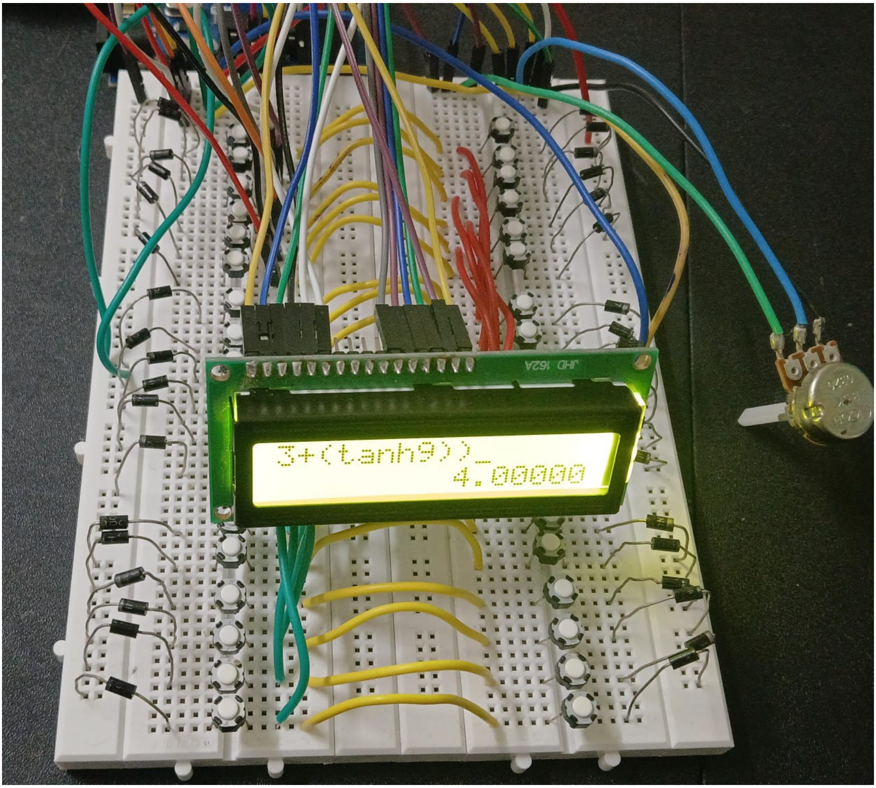


Fig. 0.1: Calculator

# 4 Mathematical Expressions

## *4.1 Core Trigonometric Functions*

- sin($x$), cos($x$), tan($x$) : Implemented using RK4 to solve the differential equation $y'' = -y$ with appropriate initial conditions. These provide the fundamental trigonometric operations.

## *4.2 Inverse Trigonometric Functions*

- arcsin($x$), arccos($x$), arctan($x$) : Calculate inverse trigonometric functions using RK4 with their respective differential equations.

## *4.3 Exponential and Logarithmic Functions*

- pow($x, w$) : Power function using RK4 for $\frac{dy}{dx} = w\frac{y}{x}$
- ln($x$) : Natural logarithm using RK4 for $\frac{dy}{dx} = \frac{1}{x}$

## *4.4 Hyperbolic Functions*

- sinh($x$), cosh($x$), tanh($x$) : Hyperbolic functions implemented using the exponential definitions.

## *4.5 Utility Functions*

- fast_inv_sqrt($x$) : Fast inverse square root using the famous Quake III algorithm.
- principal_range($\theta$) : Normalizes angles to $[0, 2\pi]$ range.
- factorial($n$) : Calculates factorial for non-negative integers.
- dtf(decimal, $*$numerator, $*$denominator) : Converts decimals to fractions.

## *4.6 Mathematical Constants*

- $\pi = 3.14159265358979323846$
- $e = 2.7182818284$

## *4.7 Important Implementation Notes*

1) The functions use a step size $H = 0.01$ for numerical approximation.
2) Error handling is implemented for edge cases (negatives, zeros).
3) The code uses fast_inv_sqrt for optimization where appropriate.

For a calculator implementation, these functions provide a complete set of mathematical operations covering:

- Basic arithmetic (through pow)
- Trigonometry (standard and inverse functions)
- Exponential and logarithmic calculations
- Hyperbolic functions
- Number theory operations (GCD, factorial, decimal-to-fraction conversion)

The numerical approach using RK4 is particularly interesting as it solves the functions through differential equations rather than series expansions, potentially offering good performance for a wide range of inputs.

## 4.8 Code-funcs.c

```c

#include <stdint.h>

#define PI 3.14159265358979323846
#define H 0.01
#define MAX_ITERS 100000
#define E 2.7182818284

double fast_inv_sqrt(double x){
    if(x <= 0) return 0;
    x = (float) x;
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = x * 0.5F;
    y = x;
    i = * ( long * ) &y;
    i = 0x5f3759df - ( i >> 1 );
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );

    return (double) y;
}

double principal_range(double angle) {
    int k = (int)(angle / (2*PI));
    angle -= k * (2*PI);
    if (angle < 0) angle += (2*PI);
    return angle;
}

// Sine calculation using RK4 for y'' = -y
double sin(double x_target) {
    double x = 0.0; // Start point
    double y = 0.0; // y(0) = 0
    double dy = 1.0; // y'(0) = 1

    double h = H; // Step size
    int steps = (int)(x_target / h); // Number of steps

    for (int i = 0; i < steps; i++) {
        double k1_y = h * dy;
        double k1_dy = h * (-y);

        double k2_y = h * (dy + 0.5 * k1_dy);
        double k2_dy = h * (-(y + 0.5 * k1_y));

        double k3_y = h * (dy + 0.5 * k2_dy);
        double k3_dy = h * (-(y + 0.5 * k2_y));

        double k4_y = h * (dy + k3_dy);
        double k4_dy = h * (-(y + k3_y));

        y += (k1_y + 2 * k2_y + 2 * k3_y + k4_y) / 6.0;
        dy += (k1_dy + 2 * k2_dy + 2 * k3_dy + k4_dy) / 6.0;

```

```
        x += h;
    }

    return y; // Return y at x_target
}

double cos(double x_target) {
    double x = 0.0; // Start point
    double y = 1.0; // y(0) = 1
    double dy = 0.0; // y'(0) = 0

    double h = H; // Step size
    int steps = (int)(x_target / h); // Number of steps

    for (int i = 0; i < steps; i++) {
        double k1_y = h * dy;
        double k1_dy = h * (-y);

        double k2_y = h * (dy + 0.5 * k1_dy);
        double k2_dy = h * (-(y + 0.5 * k1_y));

        double k3_y = h * (dy + 0.5 * k2_dy);
        double k3_dy = h * (-(y + 0.5 * k2_y));

        double k4_y = h * (dy + k3_dy);
        double k4_dy = h * (-(y + k3_y));

        y += (k1_y + 2 * k2_y + 2 * k3_y + k4_y) / 6.0;
        dy += (k1_dy + 2 * k2_dy + 2 * k3_dy + k4_dy) / 6.0;

        x += h;
    }

    return y; // Return y at x_target
}

double tan(double x){
    return sin(x)/cos(x);
}

// Power function using RK4 for dy/dx = w*y/x
double pow(double x, double w) {
    if(x < 0 && ((int) w) != w) return 0;
    if (x == 0) return 0;
    if (w == 0) return 1;

    if(x < 0) return ( ((int) w)%2 == 0 ? 1: -1)*pow(-x, w);

    double x0 = 1.0, y = 1.0;
    double target = x;
    int steps = (int)((target - x0) / H);

    for (int i = 0; i < steps; i++) {
        double k1 = H * w * y / x0;
        double k2 = H * w * (y + k1/2) / (x0 + H/2);
        double k3 = H * w * (y + k2/2) / (x0 + H/2);
        double k4 = H * w * (y + k3) / (x0 + H);

        y += (k1 + 2*k2 + 2*k3 + k4) / 6;
```

```
        x0 += H;
    }

    return y;
}

// Natural log using RK4 for dy/dx = 1/x
double ln(double x) {
    if (x <= 0) return 0.0;
    if (x < 1) return -ln(1/x);

    double x0 = 1.0, y = 0.0;
    int steps = (int)((x - x0) / H);

    for (int i = 0; i < steps; i++) {
        double k1 = H / x0;
        double k2 = H / (x0 + H/2);
        double k3 = H / (x0 + H/2);
        double k4 = H / (x0 + H);

        y += (k1 + 2*k2 + 2*k3 + k4) / 6;
        x0 += H;
    }

    return y;
}

double arctan(double x) {
    double x0 = 0.0, y = 0.0;
    int steps = x >= 0 ? (int) (x / H): (int) (-x / H);
    double step_dir = (x >= 0) ? 1 : -1;

    for (int i = 0; i < steps; i++) {
        double k1 = H / (1 + x0*x0);
        double k2 = H / (1 + (x0 + H/2)*(x0 + H/2));
        double k3 = H / (1 + (x0 + H/2)*(x0 + H/2));
        double k4 = H / (1 + (x0 + H)*(x0 + H));

        y += step_dir * (k1 + 2*k2 + 2*k3 + k4) / 6;
        x0 += step_dir * H;
    }

    return y;
}

double arcsin(double x) {
    if (x < -1 || x > 1) return 0;

    double x0 = 0.0, y = 0.0;
    int steps = x >= 0 ? (int) (x / H): (int) (-x / H);
    double step_dir = (x >= 0) ? 1 : -1;

    for (int i = 0; i < steps; i++) {
        double k1 = H * fast_inv_sqrt(1 - x0*x0);
        double k2 = H * fast_inv_sqrt(1 - (x0 + step_dir*H/2)*(x0 + step_dir*H/2));
        double k3 = H * fast_inv_sqrt(1 - (x0 + step_dir*H/2)*(x0 + step_dir*H/2));
        double k4 = H * fast_inv_sqrt(1 - (x0 + step_dir*H)*(x0 + step_dir*H));

        y += step_dir * (k1 + 2*k2 + 2*k3 + k4) / 6;
```

```
        x0 += step_dir * H;
    }

    return y;
}

double arccos(double x){
    return ((PI/2) - arcsin(x));
}

double factorial(double n) {
    if(n < 0) return 0;
    if(n == 0 || n == 1) return 1;

    double result = 1.0;

    for(double i = (int) n; i >= 2 ; i--) {
        result *= i;
    }

    return result;
}

int gcd(int a, int b) {
    int temp;

    while (b != 0) {
        temp = b;
        b = a % b;
        a = temp;
    }

    return a;
}

void dtf(double decimal, long int* numerator, long int* denominator) {
    *numerator = (int)(decimal * 1000000);
    *denominator = 1000000;

    int gcd_ = gcd(*numerator, *denominator);

    *numerator = *numerator/gcd_;
    *denominator = *denominator/gcd_;
}

double sinh(double x) {
    return (pow(E, x) - pow(E, -x)) / 2;
}

double cosh(double x) {
    return (pow(E, x) + pow(E, -x)) / 2;
}

double tanh(double x) {
    return sinh(x) / cosh(x);
}
```

## 5 Overall System Architecture

The embedded calculator is implemented on an AVR microcontroller and consists of three main components:

- Expression Parsing and Evaluation System
- User Interface and Input Handling
- Hardware Interaction Layer

## 6 Expression Parsing and Evaluation System

### 6.1 Key Data Structures

The token structure represents different types of mathematical elements:

```c
typedef struct Token {
    TokenType type; // Type of token (number, operator, function)
    TokenVal val; // Value of the token
} Token;
```

Listing 1: Token Structure

### 6.2 Shunting Yard Algorithm for Parsing

The calculator uses the Shunting Yard algorithm to convert infix expressions to Reverse Polish Notation (RPN). This algorithm ensures operator precedence and left-to-right evaluation.

The algorithm is implemented as follows:

```c
void processTokens(Token token_stream[], short size, Token output_stack[], short
      *output_size, Token operator_stack[], short *operator_size) {
    for (int i = 0; i < size; i++) {
        Token token = token_stream[i];

        if (token.type == NUM) {
            append(output_stack, output_size, token);
        } else if (token.type == OP) {
            while (*operator_size > 0 && precedence(token.val.op) <=
                  precedence(operator_stack[*operator_size - 1].val.op)) {
                append(output_stack, output_size, pop(operator_stack,
                      operator_size));
            }
            append(operator_stack, operator_size, token);
        } else if (token.type == LBRAK) {
            append(operator_stack, operator_size, token);
        } else if (token.type == RBRAK) {
            while (*operator_size > 0 && operator_stack[*operator_size - 1].type !=
                  LBRAK) {
                append(output_stack, output_size, pop(operator_stack,
                      operator_size));
            }
            pop(operator_stack, operator_size); // Remove left bracket
        }
    }

    while (*operator_size > 0) {
        append(output_stack, output_size, pop(operator_stack, operator_size));
```

```
    }
}
```

Listing 2: Shunting Yard Algorithm Implementation

### *6.3 Evaluation of RPN Expressions*

Once the expression is converted to RPN, it is evaluated using a stack-based approach.

```
double evaluateRPN(Token output_stack[], short output_size, double ans) {
    Token res_stack[STACK_SIZE];
    short res_size = 0;

    for (int i = 0; i < output_size; i++) {
        Token token = output_stack[i];

        if (token.type == NUM) {
            append(res_stack, &res_size, token);
        } else if (token.type == OP) {
            Token right = pop(res_stack, &res_size);
            Token left = pop(res_stack, &res_size);
            Token res_token;
            res_token.type = NUM;

            switch (token.val.op) {
                case ADD: res_token.val.num = left.val.num + right.val.num; break;
                case SUB: res_token.val.num = left.val.num - right.val.num; break;
                case MUL: res_token.val.num = left.val.num * right.val.num; break;
                case DIV: res_token.val.num = left.val.num / right.val.num; break;
                case POW: res_token.val.num = pow(left.val.num, right.val.num);
                    break;
            }
            append(res_stack, &res_size, res_token);
        }
    }
    return res_stack[0].val.num;
}
```

Listing 3: RPN Evaluation

**Remark:** Button matrix implementation and operator precedence are sourced from EE24BTECH11002 - $Agamjot\ Singh$

link:-hyperlink

# 7 User Interface and Input Handling

## 7.1 Input Mechanisms

The calculator employs a 6x6 matrix keypad with the following features:

- Mode switching capability
- Button debouncing mechanism
- Two button mapping arrays for Standard and Advanced modes

The keypad handling function is implemented as follows:

```
char getKeypadInput() {
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLS; col++) {
            if (isButtonPressed(row, col)) {
                return keyMap[row][col];
            }
        }
    }
    return '\0';
}
```

Listing 4: Keypad Input Handling

## 7.2 Button Matrix Scanning and Debouncing

A button matrix scanning mechanism ensures accurate key detection. The process involves:

- Setting each row LOW sequentially
- Reading column states to detect key presses
- Implementing software debouncing for stable key readings

The button matrix scanning function is implemented as follows:

```
void scanButtonMatrix() {
    for (int row = 0; row < ROWS; row++) {
        setRowLow(row);

        for (int col = 0; col < COLS; col++) {
            if (readColumn(col) == LOW) {
                debounceButton(row, col);
            }
        }

        setRowHigh(row);
    }
}
```

Listing 5: Button Matrix Scanning

## 7.3 Debouncing Mechanism

Debouncing prevents multiple unwanted detections from a single press:

```c
void debounceButton(int row, int col) {
    _delay_ms(DEBOUNCE_TIME);
    if (readColumn(col) == LOW) {
        registerKeyPress(row, col);
    }
}
```

Listing 6: Debounce Function

### 7.4 Display Management

The LCD display:

- Supports a 16x2 character display
- Handles multi-line expressions
- Converts abbreviated function names to full names
- Manages scrolling for long expressions

## 8 Hardware Interaction Layer

### 8.1 Memory Management

- Uses EEPROM for persistent storage
- Supports memory recall and storage operations
- Employs optimized memory management using fixed-size buffers

The EEPROM read function is shown below:

```c
uint8_t readEEPROM(uint16_t address) {
    while (EECR & (1 << EEPE)); // Wait for completion
    EEAR = address;
    EECR |= (1 << EERE);
    return EEDR;
}
```

Listing 7: EEPROM Read Function

### 8.2 Microcontroller Interfaces

- Direct port manipulation for LCD and keypad
- Uses AVR-specific libraries and macros
- Minimizes dynamic memory allocation

## 9 Optimization Strategies

### 9.1 Memory Optimization

- Fixed-size statically allocated arrays
- Minimal dynamic memory usage
- Efficient bit manipulation macros
- Compact token representation using a union

### 9.2 Performance Optimization

- Efficient parsing algorithm
- Minimal computational overhead
- Direct hardware register manipulation
- Predefined constants for repeated calculations

## 10 Key Design Patterns

The software architecture employs:

- **State Machine**: Used for button handling
- **Interpreter Pattern**: Applied in expression evaluation
- **Command Pattern**: Used for button mapping

## 11 Potential Improvements

Future enhancements may include:

- Advanced error handling for complex expressions
- Additional mathematical functions
- Improved memory management
- Enhanced floating-point precision handling