Arduino Scientific Calculator: Implementation and Analysis

Durgi Swaraj Sharma - EE24BTECH11018

March 24, 2025

Abstract

This report provides an in-depth explanation of the functioning of our Arduino-based scientific calculator implementation. The calculator supports advanced mathematical operations through a Shunting Yard algorithm for expression parsing, along with custom mathematical functions implemented using Euler's method. We detail both the electronic circuit design and the software architecture, focusing on the expression evaluation process, memory-efficient mathematical function implementations, and user interface optimization.

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:

- Support for basic arithmetic operations (+, -, *, /, ˆ)
- Trigonometric functions (sin, cos, tan, inverse trigonometric functions)
- Logarithmic functions (ln, log)
- Exponential operations (e^x, power function)
- Special constants (π, e)
- Parenthetical expressions with proper precedence handling

- Smart backspace functionality for error correction
- LCD display for input and result visualization
- Optimized memory usage with custom mathematical algorithms

The calculator employs the Shunting Yard algorithm for expression evaluation and custom implementations of mathematical functions using Euler's method and numerical approximations, without relying on external libraries. This approach ensures accurate calculations while maintaining a minimal memory footprint suitable for the constrained AVR microcontroller environment.

2 Hardware Components and Circuit Design

2.1 Components List

- Arduino Uno R3 (ATmega328P microcontroller)
- 16x2 LCD display JHD162A
- 8 push buttons for input
- Resistors (10kΩ pull-up resistors for buttons)
- Potentiometer (10kΩ for LCD contrast)
- Breadboard and connecting wires

2.2 Circuit Connections

The circuit follows a standard design with the following connections:

2.2.1 LCD Display Connection

The 16x2 LCD display is connected in 4-bit mode to save microcontroller pins:

- RS (Register Select) - Arduino pin 12 (PB4)
- E (Enable) - Arduino pin 11 (PB3)
- D4-D7 - Arduino digital pins 5-2 (PD5-PD2)
- LCD VSS - GND

- LCD VDD - 5V
- LCD V0 - Connected to potentiometer for contrast adjustment

2.2.2 Button Connections

Eight push buttons are connected to Arduino inputs with pull-up resistors:

- DIGIT_BTN - Arduino pin A0 (PC0): Cycles through digits 0-9
- OPEN_PAREN - Arduino pin A1 (PC1): Adds opening parenthesis
- CLOSE_PAREN - Arduino pin A2 (PC2): Adds closing parenthesis
- OP_BTN - Arduino pin A3 (PC3): Cycles through operations (+, -, *, /, ^)
- FUNC_BTN - Arduino pin A4 (PC4): Cycles through functions (sin, cos, tan, ln, etc.)
- SET_BTN - Arduino pin 7 (PD7): Commits selection to expression
- EQUALS_BTN - Arduino pin 8 (PB0): Evaluates expression
- CLEAR_BTN - Arduino pin 9 (PB1): Backspace functionality

3 Software Architecture

3.1 Code Organization

The software is structured into several logical components:

- **Core Mathematical Functions**: Custom implementations of mathematical operations without math.h
- **Hardware Initialization**: Setup of LCD, timer, and input buttons
- **User Input Handling**: Button detection and debouncing
- **Expression Building**: Construction of mathematical expressions
- **Expression Evaluation**: Shunting Yard algorithm implementation
- **Mathematical Functions**: Euler's method implementations of trigonometric, logarithmic, and exponential functions
- **Display Management**: LCD update and formatting

3.2 Key Data Structures

- **Expression String**: A character array storing the mathematical expression being built (up to 64 characters)
- **Operator Stack**: Used in the Shunting Yard algorithm to maintain proper operation precedence
- **Value Stack**: Stores operands and intermediate results during evaluation
- **Global State Variables**: Track the current input mode, selected digit/operation/function, and parenthesis depth

4 Memory-Optimized Custom Math Functions

To avoid dependency on the standard math.h library and optimize for the limited resources of the ATmega328P, we implemented custom mathematical functions:

```c
float my_abs(float x) {
    return (x < 0.0f) ? -x : x;
}

int is_zero(float x) {
    return my_abs(x) < 0.0001f;
}

int is_integer(float x) {
    float diff = x - (int)x;
    return (my_abs(diff) < 0.0001f);
}

float my_sqrt(float x) {
    if (x <= 0.0f) return 0.0f;
    float result = x;
    // Newton's method
    for (int i = 0; i < 10; i++) {
        result = 0.5f * (result + x/result);
    }
    return result;
}
```

These basic functions serve as building blocks for more complex mathematical operations and ensure consistent handling of floating-point comparisons and calculations.

5 Expression Evaluation: The Shunting Yard Algorithm

5.1 Algorithm Overview

The calculator implements Dijkstra's Shunting Yard algorithm to convert infix expressions (normal mathematical notation) to postfix notation (Reverse Polish Notation) for evaluation while respecting operator precedence.

The algorithm uses two key data structures:

1. An operator stack for operators and functions
2. A value stack for numbers and calculation results

5.2 Operator Precedence and Associativity

The calculator defines precedence levels for operations:

- Level 1: Addition (+) and subtraction (-)
- Level 2: Multiplication (*) and division (/)
- Level 3: Power (ˆ) - with right associativity
- Level 4: Functions (sin, cos, tan, ln, etc.)

This is implemented in the code through the `get_precedence` function:

```c
int get_precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;  // Higher precedence for power
        case 's':     // sin
        case 'c':       // cos
        case 't':       // tan
        case 'l':       // ln
        case 'g':       // log
        case 'a':       // asin
        case 'b':       // acos
        case 'd':       // atan
        case 'q':       // x^2
        case 'e':       // e^x
            return 4;  // Highest precedence for
               functions
        default:
            return 0;
    }
}
```

5.3 Implementation Details

The Shunting Yard algorithm implementation handles several key aspects:

5.3.1 Tokenization

The expression is processed character by character to identify numbers, operators, functions, and parentheses:

```c
// Parse numbers
if (isdigit(expr_copy[i]) || (expr_copy[i] == '.' &&
                             i+1 < strlen(expr_copy) &&
                             isdigit(expr_copy[i+1]))) {
    float number = 0.0;
    int decimal_places = 0;
    int decimal_point = 0;

    while (i < strlen(expr_copy) &&
          (isdigit(expr_copy[i]) || expr_copy[i] == '.')
            ) {
        if (expr_copy[i] == '.') {
            decimal_point = 1;
        } else {
            if (decimal_point) {
                decimal_places++;
                float decimal_value = (expr_copy[i] - '0
                   ');
                for (int j = 0; j < decimal_places; j++) {
                    decimal_value /= 10.0f;
                }
                number += decimal_value;
            } else {
                number = number * 10.0f + (expr_copy[i] - '0');
            }
        }
        i++;
    }

    value_stack[value_stack_top++] = number;
    continue;
}
```

5.3.2 Right Associativity for Power

The implementation properly handles the right-associative nature of the power operator:

```
// Pop operators with higher or equal precedence, except for right-associative ^ operator
while (operator_stack_top > 0 &&
       operator_stack[operator_stack_top - 1] != '(' &&
       ((expr_copy[i] != '^' &&
         get_precedence(operator_stack[operator_stack_top - 1]) >= get_precedence(expr_copy[i])) ||
        (expr_copy[i] == '^' &&
         get_precedence(operator_stack[operator_stack_top - 1]) > get_precedence(expr_copy[i])))) {

    char op = operator_stack[--operator_stack_top];
    // Apply the operator...
}
```

5.3.3 Function Application

The algorithm handles mathematical functions by mapping them to single-character codes:

```c
// Apply function operators
if (op == 's' || op == 'c' || op == 't' || op == 'l' ||
   op == 'g' ||
    op == 'a' || op == 'b' || op == 'd' || op == 'q' ||
       op == 'e') {

    float arg = value_stack[--value_stack_top];
    float result = 0.0f;

    switch (op) {
        case 's': { // sin
            float sin_val, cos_val;
            compute_sin_cos(arg, &sin_val, &cos_val);
            result = sin_val;
            break;
        }
        case 'c': { // cos
            float sin_val, cos_val;
            compute_sin_cos(arg, &sin_val, &cos_val);
            result = cos_val;
            break;
        }
        // Other function cases...
    }

    value_stack[value_stack_top++] = result;
}
```

6 Mathematical Functions Implementation

The calculator implements various mathematical functions without using external libraries, primarily using Euler's method for differential equations and numerical approximations.

6.1 Trigonometric Functions

Sine and cosine are implemented using Euler's method to solve the system of differential equations:

$$\frac{d\sin(x)}{dx} = \cos(x) \tag{1}$$
$$\frac{d\cos(x)}{dx} = -\sin(x) \tag{2}$$

Starting with initial conditions $\sin(0) = 0$ and $\cos(0) = 1$, the functions are numerically integrated:

```c
float compute_sin_cos(float x, float* sin_val, float* cos_val) {
    float s = 0.0f;  // sin(0)
    float c = 1.0f;  // cos(0)
    float h = 0.0001f; // smaller step size for better accuracy

    // Normalize x to $[0, 2\pi)$
    while (x >= 2*PI) x -= 2*PI;
    while (x < 0) x += 2*PI;

    for (float t = 0.0f; t < x; t += h) {
        float s_new = s + h * c;
        float c_new = c - h * s;
        s = s_new;
        c = c_new;
    }

    *sin_val = s;
    *cos_val = c;
    return s;
}
```

This approach uses the finite difference method to iteratively approximate the solutions to the differential equations. The accuracy is controlled by the step size `h` (0.0001 provides a good balance between speed and precision).

6.2 Exponential Function

The exponential function e^x is implemented by solving the differential equation:

$$\frac{dy}{dx} = y \tag{3}$$

With initial condition $y(0) = 1$, using Euler's method:

```c
float compute_exp(float x) {
    float result = 1.0f;  // e^0
    float h = 0.01f;      // step size

    if (x >= 0) {
        for (float t = 0.0f; t < x; t += h) {
            result += h * result;
        }
    } else {
        x = -x;
        for (float t = 0.0f; t < x; t += h) {
            result -= h * result;
        }
        result = 1.0f / result;
    }

    return result;
}
```

This function handles both positive and negative exponents by leveraging the relationship $e^{-x} = 1/e^x$.

6.3 Logarithmic Functions

Natural logarithm is implemented using numerical integration based on the definition:

$$\ln(x) = \int_1^x \frac{1}{t} dt \tag{4}$$

```c
float compute_ln(float x) {
    if (x <= 0) return -99999.0f; // Error value for ln(
       negative)

    float result = 0.0f;  // ln(1)
    float h = 0.001f;     // step size
    float current = 1.0f;

    if (x > 1.0f) {
        while (current < x) {
            result += h * (1.0f / current);
```

```
                current += h * current;
            }
    } else { // 0 < x < 1
            while (current > x) {
                current -= h * current;
                result -= h * (1.0f / current);
            }
    }

    return result;
}
```

Base-10 logarithm is derived using the change of base formula:

```
float compute_log10(float x) {
    // log10(x) = ln(x) / ln(10)
    return compute_ln(x) / compute_ln(10.0f);
}
```

6.4 Power Function

The power function handles integer exponents and uses the logarithm method for fractional exponents:

```
float compute_power(float base, float exponent) {
    // Special cases
    if (exponent == 0.0f) return 1.0f;
    if (base == 0.0f) return 0.0f;

    // Integer exponents - use direct multiplication for
        better accuracy
    if (is_integer(exponent)) {
        if (exponent > 0) {
            float result = 1.0f;
            for(int i = 0; i < (int)exponent; i++) {
                result *= base;
            }
            return result;
        } else {
            float result = 1.0f;
            for(int i = 0; i < (int)-exponent; i++) {
                result *= base;
            }
            return 1.0f / result;
        }
    }

    // For fractional exponents: a^b = e^(b*ln(a))
    return compute_exp(exponent * compute_ln(base));
}
```

Unfortunately, this implementation ultimately failed to provide the needed accuracy.

6.5 Inverse Trigonometric Functions

Inverse trigonometric functions are implemented using binary search and numerical integration methods:

```
float compute_asin(float x) {
    // Make sure x is in valid range [-1, 1]
    if (x < -1.0f) x = -1.0f;
    if (x > 1.0f) x = 1.0f;

    // Special cases
    if (x == 0.0f) return 0.0f;
    if (x == 1.0f) return PI/2;
    if (x == -1.0f) return -PI/2;

    // Binary search between -PI/2 and PI/2
    float low = -PI/2;
    float high = PI/2;
    float mid, sin_mid, cos_mid;

    for (int i = 0; i < 20; i++) { // 20 iterations for
       precision
        mid = (low + high) / 2;
        compute_sin_cos(mid, &sin_mid, &cos_mid);

        if (sin_mid < x) {
            low = mid;
        } else {
            high = mid;
        }
    }

    return (low + high) / 2;
}
```

7 User Interface and Interaction Flow

7.1 Input Method

The calculator uses a cycling input method to maximize functionality with minimal buttons:

- Digit button cycles through digits 0-9
- Operation button cycles through +, -, *, /, ^
- Function button cycles through 13 available functions:
  - sin, cos, tan (trigonometric functions)
  - x^2, e^x (power functions)
  - ln, log (logarithmic functions)
  - e, π (constants)
  - decimal point (.)
  - $arcsin$, $arccos$, $arctan$ (inverse trigonometric functions)
- SET button commits current selection to the expression

This implementation is handled through the `input_mode` variable:

```
// Check digit button (cycles through 0-9)
if (!(PINC & (1 << DIGIT_BTN_PIN))) {
    current_digit = (current_digit + 1) % 10;
    input_mode = 0;
    update_display();
    last_button_time = milliseconds;
}

// Check operation button (cycles through +, -, *, /, ^)
if (!(PINC & (1 << OP_BTN_PIN))) {
    current_operation = (current_operation + 1) % 5;
    input_mode = 1;
    update_display();
    last_button_time = milliseconds;
}

// Check function button (cycles through all functions)
if (!(PINC & (1 << FUNC_BTN_PIN))) {
    current_function = (current_function + 1) % 13;
```

```c
    input_mode = 2;
    update_display();
    last_button_time = milliseconds;
}
```

7.2 Display Feedback

The 16x2 LCD display provides informative feedback during expression building:

- First line: Current expression being built
- Second line: Current selection (digit, operation, or function)
- Parenthesis depth counter to help with proper expression nesting

```c
// Update LCD display based on calculator state
void update_display(void) {
    lcd_clear();
    char buffer[17];

    // First line: Show expression
    if (expr_index > 0) {
        // If expression is longer than 16 chars, show
           the last 16
        if (expr_index > 16) {
            lcd_string(&expression[expr_index - 16]);
        } else {
            lcd_string(expression);
        }
    } else {
        lcd_string("0");
    }

    // Second line: Show current selection based on
       input mode
    lcd_position(1, 0);
    switch(input_mode) {
        case 0: // Digit mode
            sprintf(buffer, "Digit: %d", current_digit);
            lcd_string(buffer);
            break;
        case 1: // Operation mode
```

```c
                sprintf(buffer, "Op: %c", operation_symbols[current_operation]);
                lcd_string(buffer);
                break;
            case 2: // Function mode
                sprintf(buffer, "Func: %s", function_names[current_function]);
                lcd_string(buffer);
                break;
    }

    // Show parenthesis depth if any are open
    if (parenthesis_depth > 0) {
        lcd_position(1, 10);
        sprintf(buffer, "P:%d", parenthesis_depth);
        lcd_string(buffer);
    }
}
```

7.3 Smart Backspace Functionality

The calculator implements an intelligent backspace feature that understands expression syntax:

```c
void backspace_expression(void) {
    if (expr_index > 0) {
        // Check if we're removing a decimal point
        if (expression[expr_index - 1] == '.') {
            decimal_entered = 0;
        }

        // Check if last character is closing parenthesis
        if (expression[expr_index - 1] == ')') {
            parenthesis_depth++;
        }
        // Check if last character is opening parenthesis
        else if (expression[expr_index - 1] == '(') {
            parenthesis_depth--;

            // Check if there's a function name before the parenthesis
            const char* func_names[] = {"sin", "cos", "tan", "ln", "log",
                                       "sin^-1", "cos^-1", "tan^-1"};
            for (int i = 0; i < 8; i++) {
                int func_len = strlen(func_names[i]);
                if (expr_index >= func_len + 1 &&
                    strncmp(&expression[expr_index - func_len - 1],
                            func_names[i], func_len) == 0) {
                    // Remove the whole function name
                    expr_index -= func_len;
                    break;
                }
            }

            // Check for x^2 and e^x separately
            if (expr_index >= 4 &&
                strncmp(&expression[expr_index - 4], "x^2", 3) == 0) {
                expr_index -= 3;
            }
            else if (expr_index >= 4 &&
                    strncmp(&expression[expr_index - 4], "e^x", 3) == 0) {
                expr_index -= 3;
            }
        }

        // Remove last character
        expr_index--;
        expression[expr_index] = '\0';
    }
}
```

This feature ensures that function names are removed in their entirety and parenthesis depth is correctly tracked when using backspace.

8 Memory Management and Optimization

8.1 Stack Usage

The calculator optimizes memory usage by using fixed-size arrays for its stacks rather than dynamic memory allocation, which would be problematic on the AVR architecture:

```
// Stacks for Shunting Yard algorithm
float value_stack[MAX_EXPR_LEN];
int value_stack_top = 0;

char operator_stack[MAX_EXPR_LEN];
int operator_stack_top = 0;
```

8.2 Debouncing Strategy

To handle button presses reliably without using interrupts for each button, the calculator implements a time-based debouncing strategy:

```
// Check if enough time has passed since last button
   press (debouncing)
if (milliseconds - last_button_time > DEBOUNCE_DELAY) {
    // Button handling code
    last_button_time = milliseconds;
}
```

This approach uses a single timer interrupt for timing rather than multiple button interrupts, saving resources.

9 Conclusion

The Arduino scientific calculator demonstrates the implementation of complex mathematical algorithms on limited hardware. The use of the Shunting Yard algorithm enables proper handling of operator precedence, while custom implementations of mathematical functions allow for advanced calculations without external libraries.

Key achievements of this project include:

- Efficient expression parsing and evaluation using the Shunting Yard algorithm

- Memory-optimized implementation of mathematical functions using numerical methods
- Support for a wide range of functions including trigonometric, logarithmic, and exponential
- Proper handling of complex expressions with nested parentheses
- Intuitive user interface despite limited input capability

Future improvements could include:

- Addition of more mathematical functions
- Memory functionality for storing results
- Improved display with scrolling for longer expressions
- Enhanced precision for floating-point calculations

This project showcases the power of implementing standard mathematical and computer science algorithms in a resource-constrained environment, providing practical experience in embedded systems programming, numerical methods, and user interface design.