Scientific Calculator using Arduino UNO and LCD

S. Sai Akshita - EE24BTECH11054

March 24, 2025

1 Introduction

This document provides an in-depth explanation of the Scientific Calculator implemented using an Arduino UNO, a Liquid Crystal Display(LCD), 20+ push buttons, and other required components.

2 Components Used

- Arduino - 1
- Breadboard - 2
- 16x2 LCD Display (Parallel, 16-pin, Non-I2C)
- USB A to USB B cable - 1
- OTG adapter - 1
- Jumper wires (Male-Male) - 50 to 70
- Potentiometer/Resistors -15.000 Ω − 6and 1.000 Ω − 2 (used in this)
- Push Buttons - 20 to 25

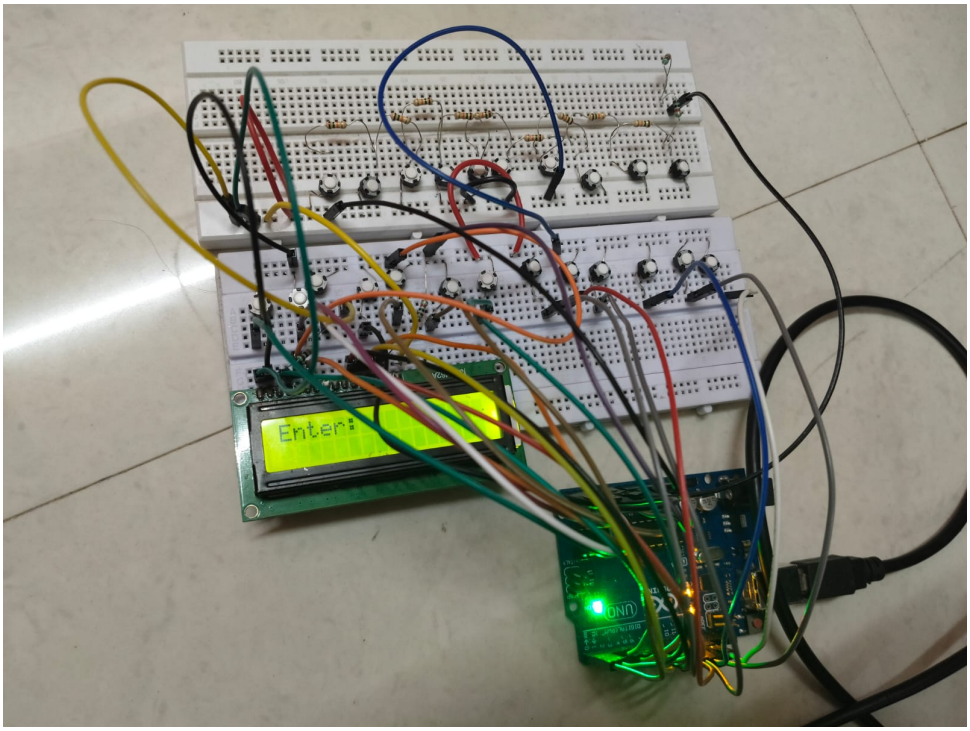

Figure 0: Scientific Calculator Setup

3 Pin Configuration and Breadboard Connections

3.1 LCD pin configuration

Pin	Connection	Description
VSS	GND	Ground
VDD	5V	Power Supply
V0	10k Potentiometer (middle pin)	Contrast Adjustment
RS	D2	Register Select
RW	GND	Read/Write (set to Write)
E	D3	Enable
D4	D4	Data Bit 4
D5	D5	Data Bit 5
D6	D6	Data Bit 6
D7	D7	Data Bit 7
A (LED+)	5V with a Resistor	Backlight Power
K (LED-)	GND	Backlight Ground

4 Button Connections for Scientific Calculator

The following list outlines the button connections for the scientific calculator:

Functionality of buttons	Corresponding Arduino Connection
Digit 0-9 Selection	A0 (Analog Read)
Addition (+)	Digital Pin 13
Subtraction (-)	Digital Pin 12
Multiplication (*)	Digital Pin 11
Division (/)	Digital Pin 10
Open Bracket (()	Digital Pin 9
Universal Constants	A1 (Analog Read)
trigonometric functions ($\sin(x), \cos(x), \tan(x)$)	A2 (Analog Read)
Close Bracket ())	A3 (Analog Read)
Logarithm (ln/log)	A4 (Analog Read)
Clear	A5(Analog Read)
Evaluate (Calculate/Enter)	Digital Pin 8
Decimal Point (.)	Digital Pin 1
Delete (Backspace)	Digital Pin 0

5 Power Supply Connection and Uploading Code

The Arduino UNO is powered via USB from a mobile phone.

- To compile the code and upload it into arduino UNO, run the command ”make” in the directory containing .c file, header file, file containing definitions and Makefile. Then the created .hex file should be uploaded into Arduino UNO.

6 Circuit Working Mechanism(Analog to Digital Conversion)

Analog-to-Digital Conversion (ADC) is a technique used to convert an analog signal into a digital value that can be processed by a microcontroller.

6.1 ADC Fundamentals

The ATmega328P’s 10-bit ADC converts analog voltages (0-5V) to digital values (0-1023). Key characteristics include:

- 10-bit resolution (1024 discrete values)
- 9-260μs conversion time (depending on clock prescaler)
- 8 multiplexed input channels (A0-A7)

6.2 Implementation in Calculator

- **Button Matrix:**
  - Digit buttons 0-9 connected via voltage divider to A0

  - Each button produces a unique voltage level (e.g., Button1=0.5V, Button2=1.0V,...,Button9=4.5V,B

- **Configuration:**
  - Reference voltage: 5V
  - Prescaler: 128 (ADC clock = 125kHz)
  - Channel selection: ADC0 (A0 pin)
- **Reading Process:**
  - Single conversion initiated via ADSC bit
  - `Conversion complete` flag (ADIF) checked for completion
  - Result read from ADC/ADCL registers

6.3 Advantages of ADC for Button Inputs

The use of ADC for button inputs offers several benefits:

- **Optimized Resource Utilization:** Reduces the number of required input pins, allowing efficient microcontroller resource management.
- **Simplified Circuit Design:** Minimizes wiring complexity, thereby improving circuit reliability.
- **Accurate and Efficient Detection:** Utilizes predefined voltage thresholds to ensure precise button identification.

6.4 Application in the Scientific Calculator

In this project, the ADC technique is used to read multiple buttons through a single analog pin, preserving valuable digital input pins. Arithmetic operations are managed using separate digital pins, while trigonometric functions are accessed using a single button with a scrolling selection mechanism.

7 Mechanism of Codes

The main.c program is located in the directory named `codes`. Below is an overview of the essential functions used in the implementation:

7.1 Core Mathematical Functions

7.1.1 Angle Conversion and Reduction

- `double reduce_angle(double rad)`: Adjusts any angle to the range $[0, 2\pi]$ by eliminating full rotations.
- `double deg2rad(double deg)`: Converts degrees into radians using the formula $rad = deg \times \frac{\pi}{180}$.

7.1.2 Natural Logarithm Calculation

- `double compute_ln(double x)`: Computes the natural logarithm by numerically integrating $\int_1^x \frac{1}{t} dt$ using the trapezoidal rule.

7.1.3 Numerical Methods

- `double tangent_rk4(double radians, double h)`: Determines the tangent using the 4th-order Runge-Kutta method by solving the differential equation $\frac{dy}{dx} = -y$.
- `double power(double x, double n)`: Computes x^n using Euler's method for the differential equation $\frac{dy}{dx} = n \cdot y$.

7.2 Stack Operations

7.2.1 Operator Stack

- `void initStack(Stack *s)`: Initializes the operator stack.
- `void push(Stack *s, const char* val)`: Adds an operator to the stack.
- `const char* peek(Stack *s)`: Retrieves the top element without removing it.

7.2.2 Number Stack

- `void initNumStack(NumStack *s)`: Initializes the number stack.
- `void pushNum(NumStack *s, float val)`: Pushes a number onto the stack.
- `float popNum(NumStack *s)`: Removes and returns the top number from the stack.

7.3 Expression Processing

7.3.1 Infix to Postfix Conversion

- `void infixToPostfix(const char* infix, char* postfix)`: Converts an expression from standard mathematical notation to Reverse Polish Notation using Dijkstra's shunting-yard algorithm.
- This function handles:
  - Parentheses grouping
  - Operator precedence (PEMDAS rules)
  - Special functions such as `trig` and `log`

7.3.2 Postfix Evaluation

- `float evaluatePostfix(const char* postfix)`: Evaluates expressions written in Reverse Polish Notation using a number stack.
- It supports:
  - Basic arithmetic operations ($+, -, \times, \div$)

- Exponentiation (x^y)
- Trigonometric functions ($\sin, \cos, \tan$)
- Logarithmic functions ($\ln, \log_{10}$)

8 Advantages of Using AVR-GCC Over C++

Using AVR-GCC instead of C++ for implementing the digital clock provides several benefits:

- **Efficiency:** AVR-GCC generates highly optimized machine code, resulting in faster execution and reduced memory usage.
- **Precise Timing:** Direct control over hardware registers allows for more accurate timing, crucial for maintaining a stable clock display.
- **Lower Overhead:** Unlike C++ with its runtime overhead, AVR-GCC offers minimal abstraction, reducing unnecessary processing load.
- **Fine-Grained Hardware Control:** Allows direct manipulation of registers and ports, ensuring precise control over multiplexing and display updates.
- **Smaller Code Size:** Since AVR-GCC avoids C++ features like classes and dynamic memory allocation, the compiled code size remains smaller.
- **Better Debugging:** With direct register access, debugging at the hardware level becomes more transparent compared to C++ abstractions.

9 Precautions

9.1 Hardware

- **Proper Power Supply:** Ensure a stable 5V power supply to prevent voltage fluctuations that can damage components.
- **Secure Connections:** Use firm jumper wire connections to avoid loose connections that may cause erratic behavior.
- **Resistor Selection:** Choose appropriate resistors, especially for LCD contrast adjustment and button input voltage dividers.
- **Debouncing Buttons:** Use pull-down or pull-up resistors to prevent unintended multiple inputs due to switch bouncing.
- **Proper LCD Handling:** Avoid excessive pressure on the LCD screen and ensure correct pin connections to prevent damage.
- **Short Circuit Prevention:** Double-check wiring to avoid accidental short circuits, especially on a breadboard.
- **Heat Dissipation:** Ensure that components, especially the Arduino, do not overheat due to excessive current draw.

- **Correct Pin Assignments:** Verify connections between buttons, LCD, and microcontroller to avoid miswiring.
- **Static Protection:** Handle microcontroller and ICs with care to prevent static discharge damage.
- **Proper Grounding:** Ensure a common ground for all components to avoid floating voltage issues.

9.2 Software

- **Efficient Code Optimization:** Keep the AVR-GCC code optimized to avoid unnecessary delays in button response.
- **Debounce Logic in Software:** Implement software-based debounce techniques to prevent unintended button presses.
- **Memory Management:** Avoid excessive memory usage to prevent crashes or unexpected behavior.
- **Error Handling:** Implement proper error detection and handling in calculations to prevent incorrect results.
- **Avoid Infinite Loops:** Ensure that the program does not get stuck in infinite loops due to misconfigured logic.
- **Correct Timing Functions:** Use appropriate delay functions for LCD updates without affecting button responsiveness.
- **Power Efficiency:** Optimize code to minimize power consumption, especially if using battery power.
- **Proper LCD Commands:** Use the correct initialization commands for the LCD to prevent display glitches.
- **Testing in Stages:** Test each component separately (LCD, buttons, arithmetic operations) before full integration.
- **Backup Code:** Always keep a backup of your working code before making major modifications.