Lab 3 Report: Microcontroller Calculator Implementation

Group Members:
Fares Trad (Student ID: 217281379)
Kanwarjot Singh (Student ID: 214284269)
Josh Keppo (Student ID: 210971752)
Kyle Williamson (Student ID: 218953901)

1. Overview

This lab focused on designing and implementing a functional calculator using the Raspberry Pi Pico microcontroller. The calculator performs basic arithmetic operations (addition and multiplication) and interacts with users through a simulated keypad and 7-segment display. The implementation follows a state machine design to manage input processing, calculation, and output display.

Objectives

- Create a calculator that accepts numerical inputs and arithmetic operations
- Implement integer addition and multiplication
- Display results on a 7-segment LED display
- Handle edge cases like overflow gracefully
- Develop comprehensive tests to validate functionality

2. Implementation Details

2.1 Development Process

The calculator was implemented using a systematic approach:

1. **Design Phase**: Created a state machine design to manage the calculator's operation flow
2. **Core Logic Implementation**: Implemented the arithmetic operations and input processing
3. **Hardware Interface**: Developed code to interface with keypad and 7-segment display
4. **Testing Framework**: Created a testing framework to validate functionality without physical hardware
5. **Integration & Validation**: Combined components and performed end-to-end testing

2.2 System Architecture

The calculator implementation uses a state machine with the following states:

- **IDLE**: Initial state, waiting for first input
- **FIRST_NUM**: Entering the first operand
- **OP_SELECTED**: Operator selected, waiting for second operand
- **SECOND_NUM**: Entering the second operand
- **RESULT**: Displaying calculation result
- **ERROR**: Error state (overflow, etc.)

2.3 Key Implementation Decisions

- **State Machine Design**: Used a state-based approach to manage calculator flow, making the code more maintainable and easier to debug
- **Hardware Abstraction**: Created a clean separation between core logic and hardware interfaces to enable testing without physical components
- **Modular Functions**: Implemented distinct functions for keypad scanning, display refresh, and arithmetic operations
- **Error Handling**: Added overflow detection and graceful error display

3. Code Explanation

3.1 Key Components

The implementation consists of several key files:

- **lab3calculator.h**: Header file defining data structures, constants, and function prototypes
- **lab3calculator.c**: Main implementation with calculator logic and hardware interface
- **lab3calculator_test.c**: Test framework for validating calculator functionality

3.2 State Machine Implementation

```
void processKey(char key) {
   // Numbers 0-9
   if (key >= '0' && key <= '9') {
      int digit = key - '0';

      switch (currentState) {
         case IDLE:
            firstNumber = digit;
            currentState = FIRST_NUM;
            break;
         case FIRST_NUM:
            // Append digit to first number
            firstNumber = firstNumber * 10 + digit;
            // Check for overflow
            if (firstNumber > 9999) {
               currentState = ERROR;
```

```
                setErrorDisplay();
            }
            break;
        // Other states...
    }

    // Update display for appropriate states
    if (currentState == FIRST_NUM) {
        setDisplayNumber(firstNumber);
    } else if (currentState == SECOND_NUM) {
        setDisplayNumber(secondNumber);
    }
  }
  // Operation keys and other handling...
}
```

This function is central to the calculator's operation, handling incoming keypress events and updating the calculator state accordingly.

3.3 Calculation Logic

```
void calculateResult() {
  switch (currentOperator) {
    case OP_ADD:
      result = firstNumber + secondNumber;
      break;
    case OP_MULTIPLY:
      result = firstNumber * secondNumber;
      break;
    default:
      result = 0;
      break;
  }

  // Check for overflow
  if (result > 9999) {
    currentState = ERROR;
    setErrorDisplay();
  }
}
```

The calculation function performs arithmetic based on the selected operator and checks for overflow conditions.

3.4 Display Management

The calculator uses a multiplexed 7-segment display system:

```
void refreshDisplay() {
    // Update each digit sequentially
    for (int digit = 0; digit < NUM_DIGITS; digit++) {
        // Turn off all digits first
        for (int i = 0; i < NUM_DIGITS; i++) {
            gpio_put(DIGIT_PINS[i], 1);
        }

        // Skip if this position is blank
        if (digitValuesToDisplay[digit] == -1) {
            continue;
        }

        // Get the pattern for this digit
        uint8_t pattern = SEGMENT_PATTERNS[digitValuesToDisplay[digit]];

        // Set segment pins
        for (int segment = 0; segment < 7; segment++) {
            gpio_put(SEG_PINS[segment], (pattern >> (6 - segment)) & 0x01);
        }

        // Turn on this digit
        gpio_put(DIGIT_PINS[digit], 0);

        // Small delay to make digit visible
        sleep_ms(2);
    }
}
```

This function uses time-division multiplexing to display different digits on the 7-segment display.

4. Hardware & Software Configuration

4.1 Hardware Setup

The calculator is designed to work with the following hardware configuration:

- **Microcontroller**: Raspberry Pi Pico (RP2040)
- **Input Device**: 4x4 Matrix Keypad
- **Output Device**: 4-digit 7-segment LED display (common cathode)
- **Pin Assignments**:
    - Keypad rows: GPIO pins 2, 3, 4, 5
    - Keypad columns: GPIO pins 6, 7, 8, 9
    - 7-segment display segments (A-G): GPIO pins 10-16
    - 7-segment display digit select: GPIO pins 17-20

4.2 Development Environment

- **IDE**: Visual Studio Code with Raspberry Pi Pico extension
- **SDK**: Raspberry Pi Pico SDK (version 2.1.1)
- **Build System**: CMake
- **Programming Language**: C

4.3 Build Configuration

The project uses CMake for build configuration:

```
# Main application
add_executable(lab3calculator lab3calculator.c)
target_link_libraries(lab3calculator
        pico_stdlib
        hardware_gpio
)

# Test application
add_executable(lab3calculator_test lab3calculator_test.c lab3calculator.c)
target_compile_definitions(lab3calculator_test PRIVATE CALCULATOR_TEST_MODE=1)
```

5. Testing & Debugging

5.1 Testing Approach

Testing was performed using a custom test framework that enabled validation without physical hardware:

1. **Unit Testing**: Individual components (state machine, display logic) were tested in isolation
2. **Integration Testing**: End-to-end functionality was tested using simulated key sequences
3. **Edge Case Testing**: Special cases like overflow and error handling were verified

5.2 Test Cases

The testing framework included the following key test cases:

- **Basic Addition**: `2A3D` (2+3=5)
- **Basic Multiplication**: `4B5D` (4×5=20)
- **Multi-digit Operations**: `123B45D` (123×45=5535)
- **Overflow Detection**: `9999B9999D` (9999×9999, should trigger error)
- **Clear Functionality**: `123A45C` (should reset calculator)
- **Sequential Operations**: `5B5DA5D` (5×5=25, then 25+5=30)

5.3 Debugging Challenges

Several challenges were encountered during development:

1. **Redefinition Errors**: Initially encountered redefinition errors when including the main implementation in the test file. Resolved by using a proper header file and conditional compilation.

2. **Hardware Abstraction**: Creating an effective testing framework without physical hardware required carefully designing the code to abstract hardware interactions.

3. **State Machine Logic**: Debugging state transitions required careful analysis to ensure the calculator moved between states correctly based on user input.

5.4 Solutions Implemented

- **Proper Header Structure**: Implemented proper C header guards and conditional compilation
- **Mock Functions**: Created mock versions of hardware-dependent functions for testing
- **Test Harness**: Developed a test harness that could simulate user input sequences
- **Logging**: Added detailed logging to track state transitions and calculations

6. Conclusion & Reflection

6.1 Key Learnings

This lab provided valuable experience in several areas:

- **Embedded System Design**: Gained practical experience designing a microcontroller-based application
- **State Machine Implementation**: Learned how to effectively implement state machines for user interface management
- **Hardware Abstraction**: Developed skills in creating abstract interfaces for hardware components
- **Test-Driven Development**: Experienced the benefits of creating tests before implementing functionality

6.2 Challenges Overcome

The most significant challenges included:

- **Hardware Simulation**: Creating a testing environment that didn't require physical hardware
- **State Management**: Handling all the possible states and transitions in the calculator
- **Error Detection**: Implementing robust error checking for operations like overflow

6.3 Future Improvements

Several enhancements could be made to the calculator:

- **Additional Operations**: Implement subtraction, division, and more complex operations
- **Memory Functions**: Add memory store/recall functionality
- **LCD Display Support**: Extend the interface to support an LCD display for better user feedback
- **Input Validation**: Add more comprehensive input validation and error messaging
- **Power Management**: Implement power-saving features when the calculator is idle

6.4 Final Thoughts

This laboratory exercise provided practical experience in embedded system design, from conceptualization through implementation and testing. The state machine approach proved to be an effective way to manage the calculator's behavior, and the testing framework enabled thorough verification without requiring physical hardware. The modular design choices made the code more maintainable and adaptable for future extensions.