Arduino Digital Clock: Implementation and Analysis

Durgi Swaraj Sharma - EE24BTECH11018

March 24, 2025

Abstract

This report details the implementation of an Arduino-based digital clock that displays time in hours, minutes, and seconds format. The clock features multiplexed seven-segment displays controlled through a 7447 BCD decoder, with user inputs for time adjustment through pushbuttons. We discuss the hardware configuration, software architecture focusing on time-keeping, display multiplexing, and button debouncing techniques. The implementation emphasizes efficient port manipulation for optimal display refresh rates while maintaining minimal component count.

1 Introduction

The digital clock project implements a functional timekeeping device with the following key features:

- 12-hour time format display (hours, minutes, seconds)
- Multiplexed 6-digit seven-segment display system
- Single 7447 BCD-to-seven-segment decoder
- Button controls for:
  - Pausing/resuming time
  - Adjusting hours, minutes, and seconds
- Efficient display refresh through direct port manipulation
- Timer-based 1Hz clock with automatic time increment

The implementation employs basic embedded programming techniques including direct register manipulation, interrupt handling, and software debouncing, all optimized for the AVR microcontroller environment.

2 Hardware Components and Circuit Design

2.1 Components List

- Arduino Uno R3 (ATmega328P microcontroller)
- Six 7-segment displays (common anode)
- 7447 BCD-to-seven-segment decoder
- Four push buttons for input
- Resistors (10kΩ pull-up resistors for buttons, 220Ω current limiting resistors for displays)
- Breadboard and connecting wires

2.2 Circuit Connections

The circuit employs multiplexing to minimize component count:

2.2.1 7447 Decoder Connection

- BCD inputs (A, B, C, D) - Arduino digital pins 2-5 (PORTD 2-5)
- Each 7-segment display segment input (a-g) connected in parallel to the corresponding 7447 outputs
- Current-limiting resistors (220Ω) between the 7447 outputs and display segments

2.2.2 Display Connections

- Display common anodes connected to Arduino pins 8-13 (PORTB 0-5)
- The six displays represent (left to right):
  - Hours tens digit (Display 0)
  - Hours ones digit (Display 1)

  - Minutes tens digit (Display 2)
  - Minutes ones digit (Display 3)
  - Seconds tens digit (Display 4)
  - Seconds ones digit (Display 5)

2.2.3 Button Connections

- Four buttons connected to Arduino analog pins A0-A3 (PORTC 0-3):
  - A0: Pause/resume button
  - A1: Increment seconds button
  - A2: Increment minutes button
  - A3: Increment hours button
- Internal pull-up resistors enabled via PORTC

3 Software Architecture

3.1 Code Organization

The software is structured into several logical components:

- **Timer Initialization**: Configuration of Timer1 for 1Hz interrupts
- **Time Management**: Incrementation of hours, minutes, and seconds
- **Display Control**: Multiplexing of displays with BCD conversion
- **Input Handling**: Button detection and debouncing
- **Main Loop**: Time-to-digit conversion and display refresh

3.2 Key Data Structures

- **Time Variables**: Three volatile variables for hours, minutes, and seconds
- **BCD Lookup Table**: Conversion from decimal digits to BCD values
- **Digit Array**: Six-element array for storing separated digits
- **State Variables**: Tracks clock running state and button press history

4 Time Management Implementation

4.1 Timer Configuration

The clock uses Timer1 in CTC (Clear Timer on Compare Match) mode to generate precise 1-second interrupts:

```
void init_timer(void) {
    TCCR1A = 0;
    TCCR1B = (1 << WGM12) | (1 << CS12) | (1 << CS10);
    OCR1A = 15624;
    TIMSK1 |= (1 << OCIE1A);
}
```

The timer is configured to generate an interrupt every 1 second with a 16MHz clock frequency:

- Prescaler of 1024 (CS12 — CS10)
- Compare value of 15624 ($\frac{16{,}000{,}000}{1024} \times 1s - 1 = 15624$)

4.2 Time Increment Logic

The timer interrupt service routine (ISR) increments the time values with proper rollover handling:

```
ISR(TIMER1_COMPA_vect) {
    if(clock_running) {
        seconds++;
        if(seconds >= 60) {
            seconds = 0;
            minutes++;
            if(minutes >= 60) {
                minutes = 0;
                hours = (hours % 12) + 1;
            }
        }
    }
}
```

The time increment follows standard rollover rules:

- Seconds: 0-59, rolls over to 0 and increments minutes
- Minutes: 0-59, rolls over to 0 and increments hours

- Hours: 1-12 (12-hour format), rolls over from 12 to 1

The `clock_running` flag enables pausing of the clock when needed.

5 Display Multiplexing System

5.1 BCD Conversion

The display uses a 7447 BCD-to-seven-segment decoder, requiring BCD input:

```
const uint8_t bcd_lookup[10] = {
    0b0000, 0b0001, 0b0010, 0b0011, 0b0100,
    0b0101, 0b0110, 0b0111, 0b1000, 0b1001
};
```

This lookup table provides the 4-bit BCD representation for each decimal digit.

5.2 Multiplexing Implementation

The display multiplexing function selectively activates one display at a time while setting the appropriate BCD value:

```
void display_digit(uint8_t display, uint8_t digit) {
    // Turn off all displays
    PORTB &= ~0b00111111;

    // Set BCD value
    PORTD = (PORTD & 0b11000011) | ((bcd_lookup[digit]
       << 2) & 0b00111100);

    // Enable selected display
    PORTB |= (1 << display);
}
```

Key aspects of the multiplexing technique:

- All displays are turned off before any change to prevent ghosting
- BCD data is set on PORTD pins 2-5 using masks to preserve other bits
- Only the desired display is activated via PORTB

This approach enables using a single 7447 decoder for all six digits, significantly reducing component count.

6 Button Input Handling

6.1 Button Debouncing

The implementation includes an effective debouncing algorithm:

```
void check_buttons(void) {
    static uint8_t last_state = 0xFF;
    uint8_t current_state = PINC & 0x0F;
    uint8_t pressed = last_state & ~current_state;

    if(pressed) {
        // Debounce delay
        _delay_ms(50);
        current_state = PINC & 0x0F;
        pressed = last_state & ~current_state;
    }

    // Process button actions...

    last_state = current_state;
}
```

The debouncing strategy:

- Tracks previous button state to detect transitions
- Detects button press using bitwise operations
- Adds a short delay and re-samples to verify the button press
- Only processes verified button presses

6.2 Time Adjustment Functions

Each button provides a specific time adjustment function:

```
// Inside check_buttons()
if(pressed & 0x01) { // A0: Toggle clock
    clock_running ^= 1;
}
if(pressed & 0x02) { // A1: Inc seconds
    seconds = (seconds + 1) % 60;
}
if(pressed & 0x04) { // A2: Inc minutes
    minutes = (minutes + 1) % 60;
```

```
}
if(pressed & 0x08) { // A3: Inc hours
    hours = (hours % 12) + 1;
}
```

The button functions provide a simple yet complete interface for clock adjustment:

- A0 toggles between running and paused states (for setting time)
- A1-A3 provide increment functions for each time component
- Each increment function handles proper range limiting

7 Main Program Loop

The main loop handles digit calculation and display updates:

```
int main(void) {
    init_ports();
    init_timer();
    sei();

    uint8_t digits[6];

    while(1) {
        check_buttons();

        // Calculate display digits
        digits[0] = hours / 10;
        digits[1] = hours % 10;
        digits[2] = minutes / 10;
        digits[3] = minutes % 10;
        digits[4] = seconds / 10;
        digits[5] = seconds % 10;

        // Update displays
        for(uint8_t i = 0; i < 6; i++) {
            display_digit(i, digits[i]);
            _delay_us(1000);
        }
    }

    return 0;
```

```
}
```

This approach ensures:

- Regular checking of button inputs
- Time-to-digit conversion for all 6 displays
- Rapid cycling through displays for persistence of vision
- Consistent refresh rate with controlled timing

8 Optimizations

8.1 Port Manipulation

The implementation uses direct port manipulation rather than Arduino functions like `digitalWrite()`:

```
PORTB &= ~0b00111111; // Clear display enable bits
PORTB |= (1 << display); // Set specific display bit
```

Benefits of this approach:

- Much faster execution (5-10× faster than `digitalWrite()`)
- Predictable timing for multiplexing
- Atomic operations for display control

8.2 BCD Lookup Table

The BCD conversion uses a pre-calculated lookup table rather than computing BCD values on-the-fly:

```
const uint8_t bcd_lookup[10] = {0b0000, 0b0001, ...};
```

This optimization:

- Reduces computation time in the display loop
- Ensures constant-time BCD conversion regardless of digit value
- Uses program memory rather than RAM for constant data

9 Conclusion

The Arduino digital clock project demonstrates efficient implementation of a multiplexed display system with button-controlled time adjustment. Key achievements include:

- Efficient multiplexing with a single BCD decoder for six displays
- Precise timekeeping using timer interrupts
- Effective button debouncing for reliable user input
- Optimized port manipulation for rapid display updates
- 12-hour time format with proper rollover handling

This project provides practical insights into multiplexing, emphasizing hardware-software integration, timekeeping mechanisms, and user interface design in resource-constrained environments.