Digital Timer Project Documentation

Table of Contents

Project Overview

The Digital Timer project is a microcontroller-based device that provides time, date, and alarm functionality with a user-friendly interface. This embedded system utilizes a PIC microcontroller to manage a 12/24-hour clock with date tracking, multiple programmable alarms/events, and an intuitive menu-driven interface with a character LCD for display.

The system is designed to maintain accurate time using a real-time clock (RTC) chip (DS1307), store multiple alarm events in non-volatile EEPROM memory, and provide visual and audio feedback through the LCD display and alarm output.

This project demonstrates proficiency in embedded systems programming, state machine design, peripheral interfacing, and user interface development.

Key Features

- **Real-time Clock**: Maintains accurate time and date using DS1307 RTC
- **Dual Time Format**: Supports both 12-hour (AM/PM) and 24-hour time formats
- **Date Tracking**: Displays and tracks day, month, and year
- **Alarm System**: Stores and manages multiple alarm events
- **Non-volatile Storage**: Stores alarms and settings in EEPROM

- **Menu-driven Interface**: Intuitive navigation through system functions
- **User Controls**: Matrix keypad for input and control
- **Visual Feedback**: 16x2 character LCD display with clear indicators
- **Automatic Alarm Sorting**: Events are automatically organized by time
- **Alarm Indication**: Visual/audible alarm signals

Hardware Components

- **Microcontroller**: PIC microcontroller (specific model not indicated in code)
- **Display**: 16x2 Character LCD
- **Real-time Clock**: DS1307 RTC chip
- **Input Interface**: Matrix keypad with multiple switches
- **Non-volatile Memory**: External EEPROM
- **Output Indicator**: Alarm output (connected to RB0)
- **I2C Communication**: For RTC and EEPROM interaction

System Architecture

The system is structured as a finite state machine with several distinct operational states:

1. **Default State**: Displays current time, date, and next alarm
2. **Menu State**: Provides top-level navigation options
3. **Set/View Event**: Submenu for alarm management
4. **Set Time/Date**: Submenu for system time settings
5. **Set Time State**: Interface for time adjustment
6. **Set Date State**: Interface for date adjustment
7. **View Event State**: Browse stored alarms
8. **Set Event State**: Create and store new alarms

The architecture follows a modular design, with each state's functionality encapsulated in dedicated code files, promoting maintainability and clarity.

File Structure and Functionality

1. **main.c**
    - Entry point and main program loop
    - State machine dispatcher
    - System initialization
    - Time/date retrieval and formatting
2. **default_screen.c**
    - Implements the default display state

- Monitors for upcoming alarms
- Compares current time with stored events

3. **menu.c**
    - Top-level menu implementation
    - Navigation between main system functions
4. **set_view_event.c**
    - Submenu for alarm management options
5. **set_time_date.c**
    - Submenu for time and date settings
6. **set_time.c**
    - Time adjustment interface
    - 12-hour time format with AM/PM toggle
7. **set_date.c**
    - Date adjustment interface
    - Day, month, and year settings
8. **set_event.c**
    - Alarm creation and storage
    - Automatic sorting of alarms by time
9. **view_event.c**
    - Browse and navigate through stored alarms
10. **mkp.c**
    - Matrix keypad initialization and input handling

Program Workflow

1. **Initialization Phase**:
    - Configure peripherals (LCD, I2C, RTC, keypad)
    - Set initial system state to default
2. **Main Operation Loop**:
    - Continuously retrieve current time from RTC
    - Scan for user input from matrix keypad

- Dispatch to appropriate state handler based on current_state
- Monitor for alarm conditions in default state

3. **State Transitions**:

   - Move between states based on user input (key presses)
   - Return to default state after completing operations
   - Clear display during state transitions

4. **Alarm Management**:

   - Check for matching alarm time in default state
   - Activate alarm output (RB0) when alarm condition is met
   - Move to next alarm after current alarm is processed

State Machine Implementation

The system employs an enumerated type `State` to track the current operational mode:

```
typedef enum {
    e_default,
    e_menu,
    e_set_view_event,
    e_set_time_date,
    e_set_time,
    e_set_date,
    e_view_event,
    e_set_event
} State;
```

The main loop uses this state variable to call the appropriate handler function:

```
while (1) {
    get_time();
    key = read_matrix_keypad(STATE);

    if (current_state == e_default) {
        // Default state handling
    }
    else if (current_state == e_menu) {
        menu(key);
    }
    // Additional state handlers
}
```

Each state handler function processes user input, updates the display, and manages state transitions as needed.

Alarm and Event System

The alarm system stores events in EEPROM with the following structure:

- 6 bytes per event:
  - Bytes 0-1: Hour (2 digits)
  - Bytes 2-3: Minute (2 digits)
  - Byte 4: AM/PM indicator (A/P)
  - Byte 5: Second character of AM/PM (M)

Events are automatically sorted in chronological order when stored. The system:

1. Compares AM/PM period first
2. Then compares hour value
3. Finally compares minute value

The default screen function continuously compares the current time with the next upcoming event, activating the alarm output (RB0) when a match is found.

User Interface and Controls

The interface utilizes a matrix keypad with the following key mappings:

- **SW1**: Increment values/Move up
- **SW2**: Navigate between fields/Move down
- **SW4**: Enter/Confirm selection
- **SW5**: Exit/Return to previous state

The LCD display provides context-sensitive information:

- Default state: Shows time, date, and upcoming events
- Setting states: Displays format templates with blinking fields
- Menu states: Shows selection indicators and menu options
- View states: Presents stored data with navigation options

Technical Implementation Details

Time and Date Formatting

The system converts BCD values from the RTC to ASCII characters for display:

```
static void get_time(void) {
    clock_reg[0] = read_ds1307(HOUR_ADDR);
    clock_reg[1] = read_ds1307(MIN_ADDR);
    clock_reg[2] = read_ds1307(SEC_ADDR);

    // Format conversion for display
    time[0] = '0' + ((clock_reg[0] >> 4) & 0x01);
    time[1] = '0' + (clock_reg[0] & 0x0F);
    // ...
}
```

Blinking Effect for User Feedback

When editing values, the system creates a blinking effect for the selected field:

```
if (delay++ < 500) {
    // Display all fields
}
else {
    // Blank out selected field
    if (field == 0) {
        // Blink hours
    }
    // ...
}
```

Alarm Comparison Logic

The system uses a sophisticated comparison algorithm to check for alarm conditions:

```
// Compare AM/PM periods
if (e_ap < current_ap) {
    continue;
}
else if (e_ap > current_ap) {
    break;
}
else {
    // Compare hours
    if ((e_hr % 12) < (hour % 12)) {
        continue;
    }
    // ...
}
```

Future Enhancements

1. **Multiple Alarm Types**
    - Add categorization for different alarm types (wake-up, reminder, etc.)
    - Implement different alarm tones or patterns
2. **Recurring Alarms**
    - Implement daily, weekly, or custom recurring events
    - Add a repeat count or indefinite repeat option
3. **Backup Power System**
    - Add battery backup for RTC and settings
    - Implement power-fail detection and recovery
4. **Extended Display Options**
    - Support for larger displays with more information
    - Custom icons or graphics for different alarm types
5. **Advanced Time Features**
    - Stopwatch and countdown timer functions
    - Multiple time zone support
6. **Connectivity**
    - Add Bluetooth or WiFi for remote setting and monitoring
    - Smartphone app integration for configuration

7. **Environmental Sensors**
    - Temperature and humidity display
    - Weather prediction based on pressure trends
8. **User Profiles**
    - Multiple user settings and preferences
    - User-specific alarms and configurations
9. **Advanced Scheduling**
    - Calendar integration with special dates
    - Holiday and special event recognition
10. **Energy Efficiency**
    - Sleep modes and power management
    - Display brightness control based on ambient light

Interview Questions and Answers

Basic Project Questions

Q1: Can you explain the overall architecture of your Digital Timer project?

A1: The Digital Timer project is built around a state machine architecture with eight distinct operational states. It uses a PIC microcontroller interfaced with a DS1307 real-time clock for time-keeping, a character LCD for display, a matrix keypad for user input, and EEPROM for non-volatile storage of alarms. The system continuously monitors the current time and compares it with stored alarm events, activating an output when matches occur. The architecture is modular, with each functional component in separate files, making the system maintainable and extensible.

Q2: How does your project handle time and date management?

A2: The project uses a DS1307 RTC chip accessed via I2C communication to maintain accurate time and date information. The RTC provides BCD-encoded values for hours, minutes, seconds, day, month, and year, which are read periodically and converted to a user-friendly format for display. The system supports both 12-hour (with AM/PM indication) and 24-hour time formats. Users can adjust time and date through dedicated interfaces that validate inputs and write the updated values back to the RTC registers.

Q3: Explain the event/alarm system implementation in your project.

A3: The alarm system stores multiple time-based events in EEPROM with each event using 6 bytes of storage: 2 for hour, 2 for minutes, and 2 for AM/PM indication. Events are automatically sorted chronologically when added, considering AM/PM period first, then hour, and finally minute values. In the default state, the system continually compares the current time with the next upcoming event, activating an alarm signal when a match is found. After an alarm triggers, the system automatically advances to the next event in the queue.

Technical Implementation Questions

Q4: How do you implement the blinking field effect in your setting screens?

A4: The blinking effect is implemented using a simple counter-based approach. A delay counter increments in each function call, and when it's below a threshold (typically 500), all fields are displayed normally. When the counter exceeds this threshold but is below the maximum (typically 1000), the selected field is blanked out by displaying spaces instead of the actual values. When the counter reaches its maximum, it resets to zero, creating a continuous blinking cycle. The currently selected field is tracked with a field variable, determining which section to blank during the "off" portion of the cycle.

Q5: Explain how the matrix keypad scanning works in your system.

A5: The matrix keypad scanning uses a row-column approach. The rows are configured as outputs and columns as inputs with pull-ups enabled. The scan_key() function sequentially activates one row at a time (by setting it LOW) while keeping others inactive (HIGH), then checks each column for a LOW state, which indicates a pressed key at that row-column intersection. The function maps each detected key press to a unique switch identifier (SW1-SW12). Additionally, the read_matrix_keypad() function implements state-based detection to prevent key repetition, returning a key value only on the initial press rather than continuously while held.

Q6: How does your code ensure that alarm events are stored in chronological order?

A6: When a new alarm is created in set_event(), the system performs an ordered insertion. It reads existing alarms sequentially and compares them with the new alarm using a three-level comparison: first AM/PM period, then hour value, and finally minute value. When it finds the correct position (where the new alarm should occur before an existing one), it shifts all subsequent alarms forward in memory to create space, then inserts the new alarm at that position. This ensures that alarms are always stored in chronological order, simplifying the process of finding the next upcoming alarm.

Problem-Solving Questions

Q7: If the system needs to handle multiple simultaneous alarms at the same time, how would you modify your implementation?

A7: To handle multiple simultaneous alarms, I would modify the alarm checking logic to maintain a list of active alarms rather than just activating a single output. I would add an additional byte to each alarm record to store a unique alarm ID or type. When multiple alarms trigger at the same time, the system would record each active alarm ID in a queue or array. I'd implement a more sophisticated alarm display that cycles through all currently active alarms, showing their IDs or descriptions. Additionally, I'd add user controls to acknowledge and dismiss individual alarms from the active list rather than automatically advancing to the next event.

Q8: How would you add a snooze functionality to your alarm system?

A8: To implement a snooze function, I would:

1. Add a dedicated SNOOZE key (perhaps repurposing an existing key during alarm activation)

2. When an alarm triggers and the snooze key is pressed, temporarily store the current alarm details in a separate "snoozed alarm" buffer
3. Calculate a new time by adding the snooze duration (e.g., 5 minutes) to the current time
4. Create a temporary event with this new time and insert it into the event list using the existing chronological insertion function
5. Add a flag to the event structure to indicate snoozed alarms, allowing different visual indicators
6. Limit the number of snoozes per alarm using a counter field

Q9: If power is lost, how does your system recover and maintain alarm settings?

A9: The system uses EEPROM for non-volatile storage of alarm events, so these settings persist through power cycles. The RTC chip typically has battery backup capability which allows it to continue keeping time even when main power is lost. However, to improve recovery, I would add several enhancements:

1. Store a system configuration block in EEPROM with validity markers
2. Implement a power-on check that validates RTC operation and time reasonableness
3. Add a low-battery detection for the RTC backup battery
4. Implement a "last known state" recovery to return to the appropriate screen after power restoration
5. Add visual indicators for power loss events so users are aware time may need verification

Design and Future Enhancement Questions

Q10: How would you modify your design to support recurring alarms?

A10: To support recurring alarms, I would:

1. Extend the event data structure to include a recurrence byte with bit flags:
    - Bit 0-6: Days of week (Monday-Sunday)
    - Bit 7: Recurring flag (0=one-time, 1=recurring)
2. Modify the alarm comparison logic to check both time and recurrence pattern
3. Instead of removing an alarm after it triggers, check its recurrence pattern
4. For recurring alarms, keep them in the event list and implement a "next occurrence" calculation
5. Add a user interface for setting recurrence patterns when creating new alarms
6. Include visual indicators in the alarm display to show recurring vs. one-time events

Q11: If you were to redesign this project, what would you do differently?

A11: In a redesign, I would make several improvements:

1. Use a more structured approach with clearer separation between hardware abstraction, business logic, and user interface layers
2. Implement a proper scheduler instead of the current polling-based approach
3. Use interrupt-driven input handling for better responsiveness
4. Adopt a more memory-efficient event storage format
5. Implement a proper configuration system with settings validation
6. Add comprehensive error handling and recovery mechanisms
7. Use a more sophisticated display library with abstracted rendering

8. Improve the code's portability by better isolating hardware-specific components
9. Add logging capabilities for debugging and diagnostics
10. Implement a more flexible state machine with transition tables and event-driven architecture

Q12: How would you implement energy-saving features in this system?

A12: To improve energy efficiency, I would implement:

1. Sleep modes for the microcontroller during periods of inactivity
2. Display timeout that dims or turns off the LCD after a period without user input
3. Reduced polling frequency for tasks that don't require constant monitoring
4. Optimized I2C communication with the RTC to reduce transaction frequency
5. Ambient light sensing to automatically adjust display brightness
6. Power-aware state machine that tracks high and low-power states
7. User-configurable power profiles for different usage patterns
8. Battery level monitoring with low-power warning modes
9. Intelligent alarm prediction that wakes the system only when approaching alarm times
10. Energy usage analysis during development to identify and optimize power-hungry components

Conclusion

The Digital Timer project demonstrates a well-structured approach to embedded systems design using state machines, real-time clocks, user interfaces, and non-volatile storage. Through careful implementation of time management, event handling, and user interaction, the system provides reliable alarm functionality with an intuitive interface.

The modular architecture and clear separation of concerns make the codebase maintainable and extensible, allowing for future enhancements as outlined in this documentation. The technical solutions employed, such as chronological event sorting, field-based editing with visual feedback, and efficient time comparisons, showcase effective embedded programming practices.

This project serves as an excellent foundation for more advanced time management systems and demonstrates proficiency in microcontroller programming, peripheral interfacing, and user experience design for embedded applications.