Multi-Timer Application

What is the problem?

We need to track multiple simultaneous tasks, with each task requiring its own independent timer.

Our objective:

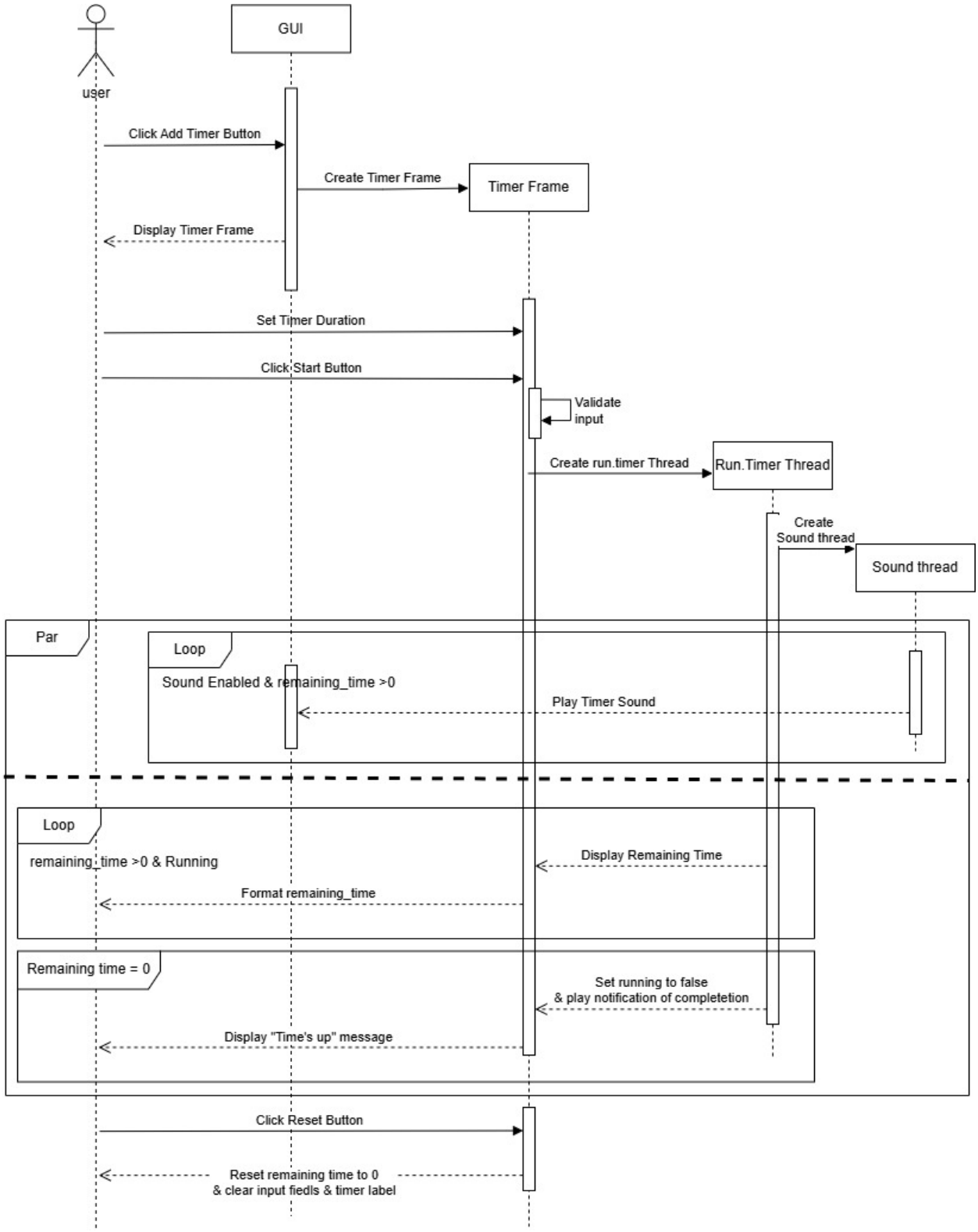
Our goal is to develop a program with multiple independent timers. Each timer should track a specific task without interfering with others. Users should have individual control over each timer—including the ability to pause, resume, and reset them.

Our solution is:

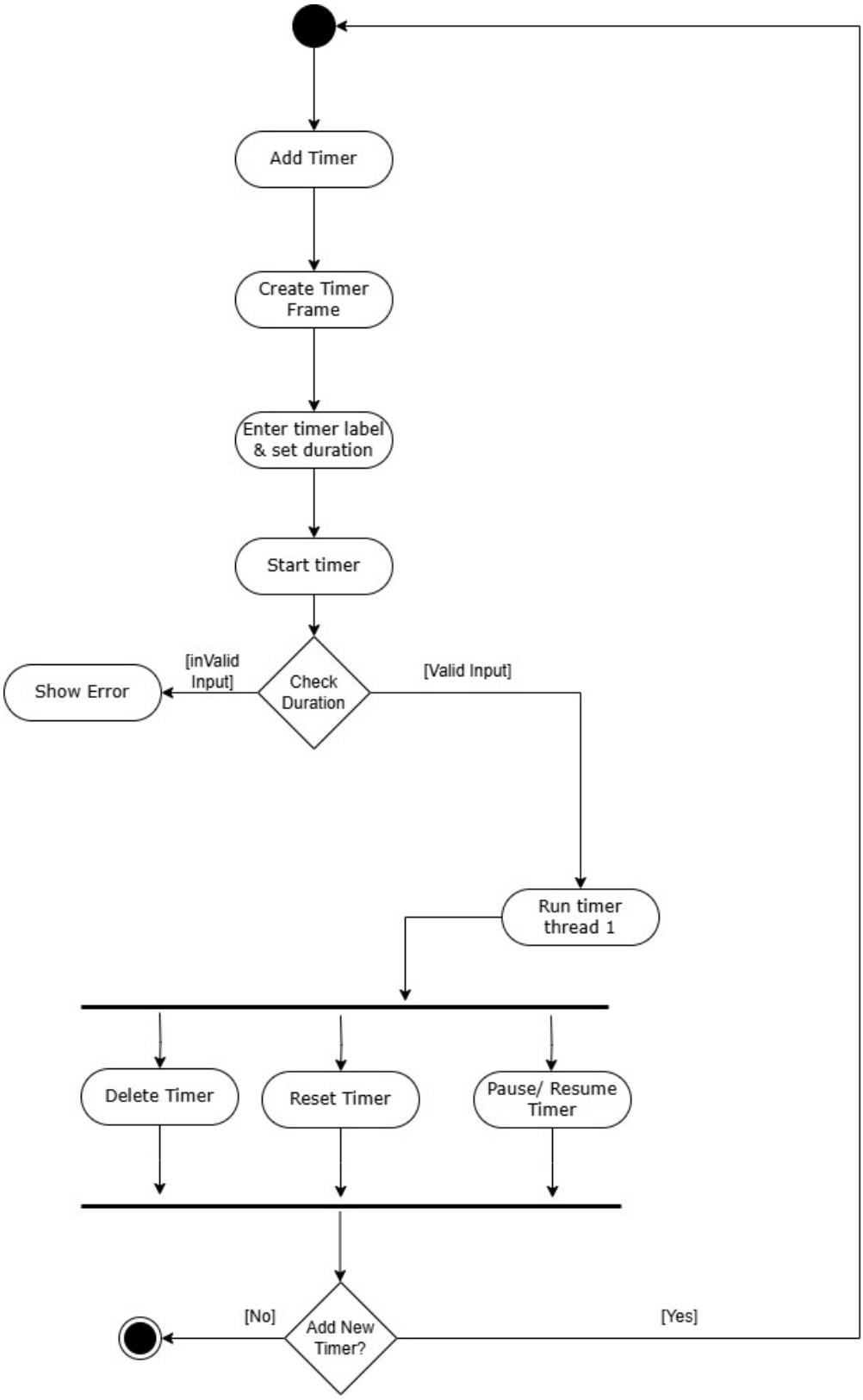
Using Python's parallel processing and threading capabilities, we created a program that runs multiple independent timers simultaneously. Each timer functions independently and includes an alarm sound. We implemented user controls for pausing, resuming, and resetting each timer.

Diagrams and explanations:

Sequence Diagram



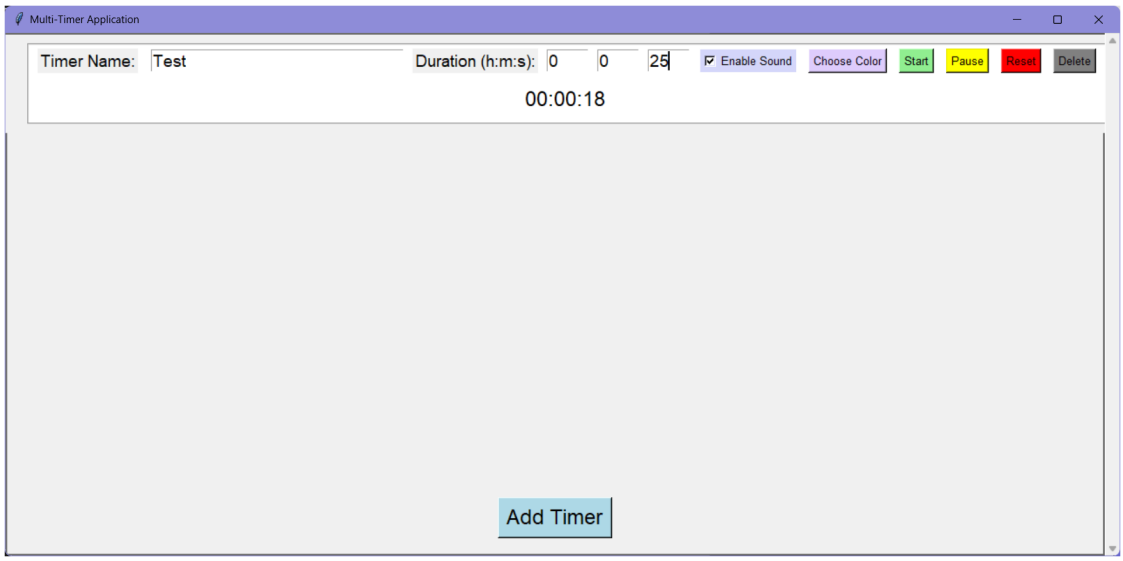
Activity Diagram


Add Timer
Create Timer Frame
Enter timer label & set duration
Start timer
Check Duration
[inValid Input]
Show Error
[Valid Input]
Run timer thread 1
Delete Timer
Reset Timer
Pause/ Resume Timer
Add New Timer?
[No]
[Yes]

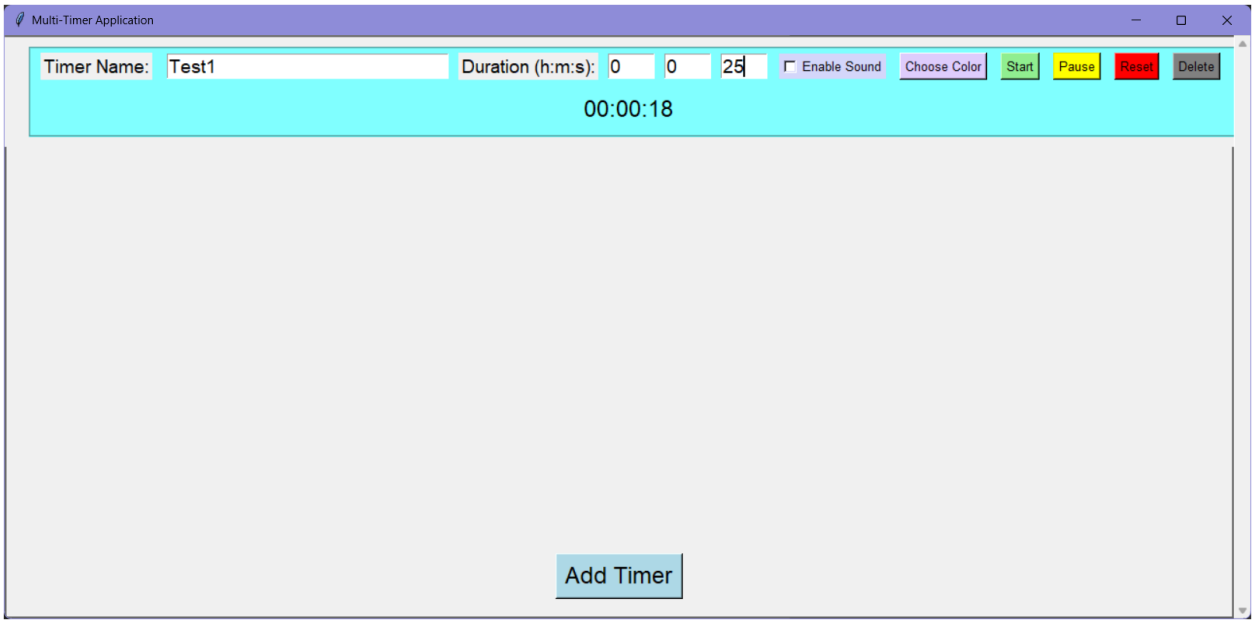
Demo:

Start App with Click “Add Timer” Button

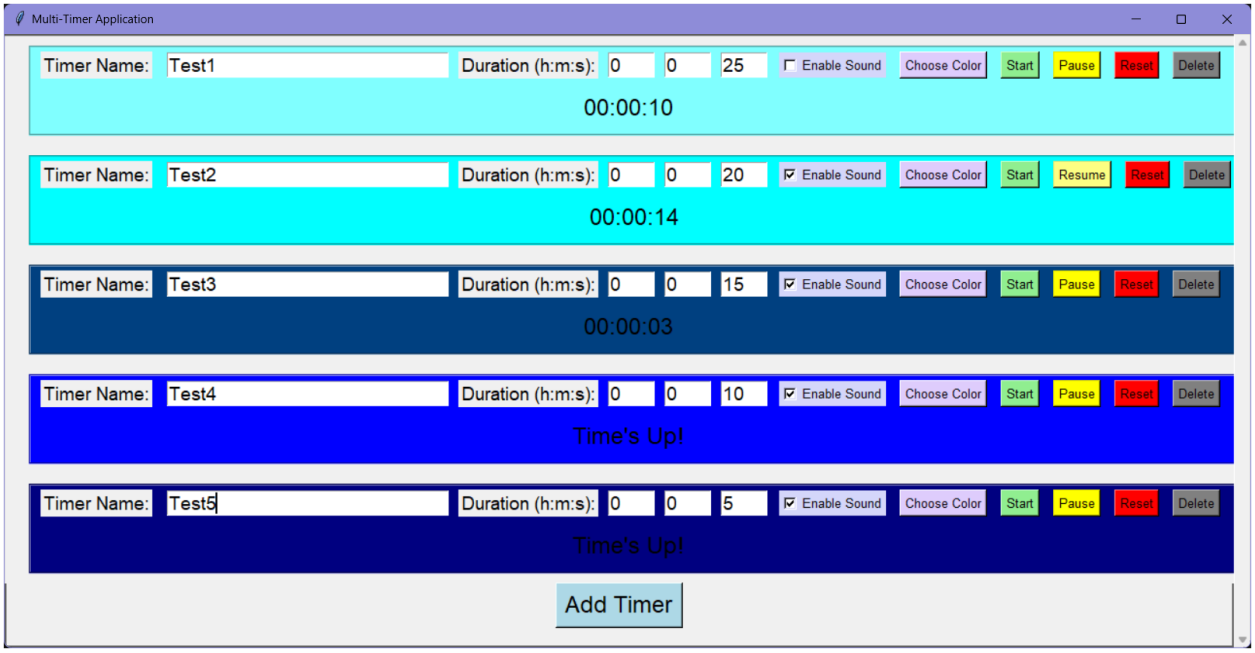
Next, Enter the Timer Label and Set Duration, Then Click “Start” Button



You can Change background color for Timer Frame,
You can “pause” and “Resume” The Timer, or Reset to Zero
OR Delete the timer frame



To add new Timer, Click “Add Timer” Button again



Now You can Start more than one Timer in the same time and this achieves the concept of parallelism

Important Libraries:

```
import tkinter as tk
from tkinter import ttk
from tkinter import colorchooser
import threading
import time
import pygame
```

Global variables

```
timers = []
global_sound_enabled = True #Default Timer Sound Effect
```

The global variables serve two main purposes in this application:

- The 'timers' list: Maintains a collection of all active timer instances in the application, allowing for tracking and managing multiple timers simultaneously
- The 'global_sound_enabled' flag: Controls the overall sound functionality for all timers, set to True by default to enable timer sound effects

These variables are essential for:

- Maintaining application state across different functions
- Managing multiple timer instances effectively
- Providing centralized sound control for all timers

play Timer sound effect

```
def play_timer_sound(timer):
    if global_sound_enabled and timer["sound_enabled"].get():
        pygame.mixer.music.load(r"D:\The college\Level 4\Paralle
        while timer["running"]:
            pygame.mixer.music.play()
            time.sleep(1)
```

This function plays a sound effect for an active timer. Here's how it works:

- Checks if both global sound and individual timer sound are enabled
- Loads the timer sound file from the specified path
- Continuously plays the sound while the timer is running
- Uses a 1-second delay between sound plays to prevent overlapping

The function runs in a separate thread to ensure the sound doesn't block the main timer functionality.

Choose color

```
def choose_color(timer):
    color = colorchooser.askcolor()[1]
    if color:
        timer["frame"].config(bg=color)
        timer["timer_label"].config(bg=color)
```

The choose_color function allows users to customize the appearance of individual timers. Here's how it works:

- Opens a color picker dialog using tkinter's colorchooser.askcolor()
- Returns a tuple containing RGB values and hex color code - we use the hex code [1]
- If a color is selected (not cancelled), updates both the timer frame and label backgrounds

- Provides visual customization to help users distinguish between multiple timers

This feature enhances the user experience by allowing visual differentiation between multiple running timers.

Complete notification

```
def notify_completion():
    pygame.mixer.music.load(r"D:\The college\Level 4\Parallel\al
    pygame.mixer.music.play()
    time.sleep(1)
    all_threads_len = threading.active_count()
    if all_threads_len > 1:
        pygame.mixer.music.load(r"D:\The college\Level 4\Paralle
```

The notify_completion function handles the timer completion notification process:

- Loads and plays a specific alarm sound when a timer completes
- Includes a 1-second delay to ensure the sound is audible
- Checks the number of active threads in the application
- If multiple timers are running (thread count > 1), reloads the regular timer sound

This function ensures users receive clear audio feedback when their timers complete while maintaining proper sound management for other active timers.

Run timer

```
def run_timer(timer):
    sound_thread = Thread(target=play_timer_sound, args=(timer,
    sound_thread.start()
    while timer["remaining_time"] > 0 and timer["running"]:
        hours, remainder = divmod(timer["remaining_time"], 3600
```

```
            minutes, seconds = divmod(remainder, 60)
            timer_format = f"{hours:02}:{minutes:02}:{seconds:02}"
            timer["timer_label"].config(text=timer_format)
            time.sleep(1)
            timer["remaining_time"] -= 1

        if timer["remaining_time"] == 0:
            timer["running"] = False
            timer["timer_label"].config(text="Time's Up!")
            notify_completion()
```

The run_timer function is a core component that handles the timer countdown process:

- Initiates a separate thread for playing the timer sound
- Continuously updates the timer display while the remaining time is greater than 0 and the timer is running
- Converts total seconds into hours, minutes, and seconds format using divmod
- Updates the timer display label every second
- Decrements the remaining time by 1 second in each iteration
- When the timer reaches zero, it:
  - Stops the timer by setting running to False
  - Updates the display to show "Time's Up!"
  - Triggers the completion notification

This function demonstrates effective use of threading to handle both the countdown and sound playing simultaneously without blocking the main application.

Pause & Resume Timer

```
def pause_timer(timer, pause_button):
    if timer["remaining_time"] > 0:
        timer["running"] = False
        pause_button.config(text="Resume", bg="#fcfe82", comman

def resume_timer(timer, pause_button):
    if timer["remaining_time"] > 0:
        timer["running"] = True
        pause_button.config(text="Pause", bg="yellow", command=
                run_thread = Thread(target=run_timer, args=(time
        run_thread.start()
```

The pause_timer and resume_timer functions work together to control timer execution:

- pause_timer:
  - Checks if there's remaining time on the timer
  - Stops the timer by setting running to False
  - Changes the button appearance and text to "Resume"
  - Updates the button's command to call resume_timer when clicked
- resume_timer:
  - Verifies there's remaining time on the timer
  - Reactivates the timer by setting running to True
  - Changes the button back to "Pause" state
  - Creates a new thread to continue the timer countdown

These functions demonstrate proper thread management and user interface updates, ensuring smooth timer control without blocking the application.

Start timer

```
def start_timer(timer, pause_button):
    if not timer["running"]:
        if timer["remaining_time"] == 0:
            hours = timer["hours_entry"].get()
            minutes = timer["minutes_entry"].get()
            seconds = timer["seconds_entry"].get()

            if not hours.isdigit() or not minutes.isdigit() or n
                raise ValueError("Please enter valid numeric val

            hours = int(hours)
            minutes = int(minutes)
            seconds = int(seconds)

            timer["remaining_time"] = hours * 3600 + minutes * 6

            if timer["remaining_time"] <= 0:
                raise ValueError("Time must be greater than zero

        timer["running"] = True
        pause_button.config(text="Pause", bg="yellow", command=l
        run_thread = Thread(target=run_timer, args=(timer,), dae
        run_thread.start()
```

The start_timer function manages the initialization and execution of individual timers:

- Checks if the timer is not already running
- When remaining time is zero, it:
  - Retrieves hours, minutes, and seconds from entry fields
  - Validates that all inputs are numeric values
  - Converts input strings to integers
  - Calculates total remaining time in seconds

    - Ensures the timer duration is greater than zero
- Once validated:
    - Sets the timer to running state
    - Updates pause button appearance and functionality
    - Creates a new daemon thread to run the timer

This function demonstrates proper input validation and thread creation for timer execution.

Reset Timer

```
def reset_timer(timer):
    timer["running"] = False
    timer["remaining_time"] = 0
    timer["timer_label"].config(text="00:00:00")
    for entry in [timer["hours_entry"], timer["minutes_entry"],
        entry.delete(0, tk.END)
        entry.insert(0, "0")
```

The reset_timer function is responsible for resetting a timer to its initial state:

- Stops the timer by setting running to False
- Clears the remaining time by setting it to 0
- Resets the timer display to "00:00:00"
- Clears and resets all time entry fields (hours, minutes, seconds) to "0"

This function ensures a clean reset of the timer, allowing users to start fresh with new time values.

Delete Timer

```
def delete_timer(timer):
    timer["running"] = False
    timer["frame"].destroy()
    timers.remove(timer)
```

The delete_timer function provides a clean way to remove timers from the application:

- Stops the timer immediately by setting running to False
- Removes the timer's visual elements from the interface using destroy()
- Removes the timer object from the global timers list

This function ensures proper cleanup of timer resources and interface elements when a timer is no longer needed.

Add timer frame

```
def add_timer(content_frame):
    timer_frame = tk.Frame(content_frame, bg="white", borderwidt
    timer_frame.pack(pady=10, padx=25, fill="x", expand=True)

    timer = {
        "frame": timer_frame,
        "running": False,
        "remaining_time": 0,
        "sound_enabled": tk.BooleanVar(value=True),
    }

    tk.Label(timer_frame, text="Timer Name:", font=("Arial", 14)
    tk.Entry(timer_frame, font=("Arial", 14), width=26).grid(row

    tk.Label(timer_frame, text="Duration (h:m:s):", font=("Arial

    timer["enter_hours"] = tk.Entry(timer_frame, font=("Arial",
    timer["enter_hours"].insert(0, "0")
    timer["enter_hours"].grid(row=0, column=3, padx=5)
```

```
    timer["enter_minutes"] = tk.Entry(timer_frame, font=("Arial'
    timer["enter_minutes"].insert(0, "0")
    timer["enter_minutes"].grid(row=0, column=4, padx=5)

    timer["enter_seconds"] = tk.Entry(timer_frame, font=("Arial'
    timer["enter_seconds"].insert(0, "0")
    timer["enter_seconds"].grid(row=0, column=5, padx=5)

    tk.Checkbutton(timer_frame, text="Enable Sound", bg="#d4d6fa

    tk.Button(timer_frame, text="Choose Color", bg="#deccfc",for

    tk.Button(timer_frame, text="Start",font=("Arial", 10), comm

    pause_button = tk.Button(timer_frame, text="Pause",font=("Ar
    pause_button.grid(row=0, column=9, padx=7)

    tk.Button(timer_frame, text="Reset",font=("Arial", 10), comm

    tk.Button(timer_frame, text="Delete",font=("Arial", 10), cor


    timer["timer_label"] = tk.Label(timer_frame, text="00:00:00'
    timer["timer_label"].grid(row=1, column=0, columnspan=12, pa

    timers.append(timer)
```

This multi-timer application is a Python-based program that enables users to manage multiple independent timers simultaneously. Here are the key components:

- **Timer Core Functions:**
    - run_timer: Handles the countdown process and updates display

    - pause_timer/resume_timer: Controls timer state and execution
    - start_timer: Initializes and begins timer countdown
    - reset_timer: Resets timer to initial state
    - delete_timer: Removes timer from application
- **User Interface Features:**
    - Individual frames for each timer
    - Input fields for hours, minutes, and seconds
    - Control buttons (Start, Pause, Reset, Delete)
    - Sound toggle option
    - Color customization
- **Technical Implementation:**
    - Uses threading for non-blocking timer operation
    - Implements proper input validation
    - Manages multiple timer states independently
    - Utilizes Tkinter for GUI components

The application provides a user-friendly interface for creating and managing multiple concurrent timers, making it suitable for tracking various timed tasks simultaneously.

Main Application

```
if __name__ == "__main__":
    root = tk.Tk()
    root.title("Multi-Timer Application")
    root.geometry("800x600")
    pygame.mixer.init()
```

```
    canvas = tk.Canvas(root)
    scrollbar = ttk.Scrollbar(root, orient="vertical", command=c
    canvas.configure(yscrollcommand=scrollbar.set)

    canvas.pack(side="left", fill="both", expand=True)
    scrollbar.pack(side="right", fill="y")

    content_frame = ttk.Frame(canvas)

    screen_width = root.winfo_screenwidth()

    canvas.create_window((0, 0), window=content_frame, anchor="n

    add_timer_button = tk.Button(canvas, text="Add Timer", comma
    add_timer_button.pack(side=tk.BOTTOM , pady=20)

    # To resize scroll region when I add many timers
    def resize_canvas(event):
        canvas.configure(scrollregion=canvas.bbox("all"))

    content_frame.bind("<Configure>", resize_canvas)

    root.mainloop()
```

The main application code establishes the core structure of the Multi-Timer Application:

- **Window Setup:**
  - Creates the main application window with specified dimensions
  - Initializes pygame mixer for sound functionality
- **Scrollable Interface:**
  - Implements a scrollable canvas with vertical scrollbar
  - Creates a content frame to hold multiple timers
  - Automatically adjusts scroll region when new timers are added

- **Controls:**
  - Adds a prominent "Add Timer" button at the bottom
  - Configures the window to be responsive to screen width

This implementation creates a flexible, user-friendly interface that can accommodate multiple timers while maintaining accessibility through scrolling functionality.