```python
def log_timestamp(event):
    global experiment_count, task_count, interval_count, event_count
    global experiment_end_count, task_end_count, interval_end_count, event_st

    if event == "Experiment Start":
        experiment_count += 1
        event_start = "- Start"
        event = f"Experiment {experiment_count} - {event_start}"
    elif event == "Experiment End":
        experiment_end_count += 1
        event_end = "- End"
        event = f"Experiment End {experiment_end_count} - {event_end}"
    elif event == "Task Start":
        task_count += 1
        event_start = "- Start"
        event = f"Task {task_count} - {event_start}"
    elif event == "Task End":
        task_end_count += 1
        event_end = "- End"
        event = f"Task End {task_end_count} - {event_end}"
```

```
    event = f"Task End {task_end_count} - {event_end}"
elif event == "Interval Start":
    interval_count += 1
    event_start = "- Start"
    event = f"Interval {interval_count} - {event_start}"
elif event == "Event Start":
    event_count += 1
    event_start = "- Start"
    event = f"Event {event_count} - {event_start}"
```

1. The function `log_timestamp` takes an `event` as input. This event represents different types of occurrences that the researcher wants to log, such as "Experiment Start," "Experiment End," "Task Start," "Task End," "Interval Start," or "Event Start."

2. Inside the function, there are global variables defined to keep track of different counts:

- `experiment_count`: Tracks the number of times the "Experiment Start" event occurs.
- `task_count`: Tracks the number of times the "Task Start" event occurs.
- `interval_count`: Tracks the number of times the "Interval Start" event occurs.
- `event_count`: Tracks the number of times the "Event Start" event occurs.
- `experiment_end_count`: Tracks the number of times the "Experiment End" event occurs.
- `task_end_count`: Tracks the number of times the "Task End" event occurs.
- `interval_end_count`: Tracks the number of times the "Interval End" event occurs.
- `event_start`: Keeps track of the occurrence of "Event Start" events.

3. The function checks the value of the `event` parameter using a series of `if-elif` statements to determine which type of event is being logged. Depending on the event type, the corresponding count is incremented, and a descriptive string is generated to represent the event with its count.

4. The `current_time` variable is set to the current date and time in the Finnish timezone ('Europe/Helsinki'). It uses the `datetime. datetime.now()` function from the `datetime` module and the `pytz.timezone` to get the current time in the desired timezone.

```
current_time = datetime.datetime.now(pytz.timezone('Europe/Helsinki')).strf
gmt_time = datetime.datetime.strptime(current_time, "%Y-%m-%d %H:%M:%S")
gmt_time = pytz.timezone("Europe/Helsinki").localize(gmt_time).astimezone(p
timestamps.append((event, gmt_time))

text.insert(END, f"{event}\t{gmt_time}\n")
```

After generating the event label, the function retrieves the current time in the "Europe/Helsinki" timezone, converts it to a GMT time, and formats it as a string. The event label and GMT time are then appended as a tuple to the timestamps list for later use. Finally, the event label and GMT time are displayed in the GUI using the text.insert method.

In summary, the log_timestamp function updates the event count, generates an event label based on the event type, converts the current time to GMT, stores the event and GMT time in the timestamps list, and displays the event and GMT time in the GUI.

```
def create_folder():
    # Declare global variables to store information related to the experime
    global experiment_name, participant_name, folder_path, file

    # Get experiment name and participant name from user input (likely thro
    experiment_name = experiment_name_entry.get()
    participant_name = participant_name_entry.get()

    # Ask the user to select the folder path where the experiment data will
    folder_path = filedialog.askdirectory()

    # Create a subfolder using the combination of the experiment name and p
    folder_name = experiment_name + "/" + participant_name
    folder_path = os.path.join(folder_path, folder_name)

    # Create the folder and ensure it's created if it doesn't exist (os.mak
    os.makedirs(folder_path, exist_ok=True)

    # Create the file name (using the participant name with .txt extension)
    file = participant_name + ".txt"
```

```
# Clear the existing entry in the Folder_path_entry widget and upda
Folder_path_entry.delete(0, END)  # Clear existing entry
Folder_path_entry.insert(0, folder_path)

# Insert information about the experiment, participant, folder path
text.insert(END, f"Experiment Name: {experiment_name}\n"
                 f"Participant Name: {participant_name}\n"
                 f"Folder Path: {folder_path}\n"
                 f"File: {file}\n\n\n"
                 f"\t\t\tTimestamp\n")

# Enable the "Experiment Start" button and disable the folder selec
experiment_start_button.config(state=tk.NORMAL)
folder_button.config(state=tk.DISABLED)

# Disable the experiment name and participant name entry fields aft
experiment_name_entry.config(state=tk.DISABLED)
participant_name_entry.config(state=tk.DISABLED)
```

Explanation:

1. The `create_folder` function sets up the data storage for the experiment by creating a specific folder structure where the experiment data will be saved.

2. It starts by declaring global variables (`experiment_name`, `participant_name`, `folder_path`, and `file`) to store the relevant information about the experiment and participant.

3. The function retrieves the experiment name and participant name from entry widgets (`experiment_name_entry` and `participant_name_entry`) where the user would have provided this information.

4. The `filedialog.askdirectory()` function opens a dialog box for the user to select the folder where they want to save the experiment data. The selected folder path is stored in the `folder_path` variable.

5. The function creates a subfolder within the selected folder path using the combination of the experiment name and participant name. This subfolder is where the experiment data for a specific participant will be saved.

6. The `os.makedirs(folder_path, exist_ok=True)` line ensures that the subfolder is created. The `exist_ok=True` argument ensures that the function does not raise an error if the folder already exists.

7. The function then generates the name of the file that will be used to save the logged data. The file name is simply the participant name with the ".txt" extension.

8. The `Folder_path_entry` widget (assuming it's an entry widget) is updated to display the selected folder path.

9. The `text` widget (probably a text widget in the GUI) is updated to display information about the experiment, participant, folder path, and file name.

10.The"ExperimentStart"buttonisenabled(`experiment_start_button.config(state=tk.NORMAL)`) to allow the user to start logging timestamps.

11. The folder selection button is disabled (`folder_button.config(state=tk.DISABLED)`) to prevent selecting a different folder after it has been created.

12. Finally, the entry fields for the experiment name and participant name are disabled (`experiment_name_entry.config(state=tk.DISABLED)`and `participant_name_entry.config(state=tk.DISABLED)`) to prevent changing the values after the folder has been created.

Overall, the `create_folder` function provides a way for the user to set up the experiment data storage by selecting a folder and creating a subfolder for each participant. It also updates the GUI to display the relevant information and enables/disables appropriate buttons and fields to guide the user through the experiment setup process.

```
def start_experiment():
    # Declare global variables for the buttons
    global experiment_start_button, experiment_end_button, interval_start_butt

    # Disable the "Experiment Start" button
    experiment_start_button.config(state=tk.DISABLED)

    # Enable the buttons for logging experiment end, interval start, task sta
    experiment_end_button.config(state=tk.NORMAL)
    interval_start_button.config(state=tk.NORMAL)
    task_start_button.config(state=tk.NORMAL)
    event_start_button.config(state=tk.NORMAL)

    # Log the timestamp for the experiment start
    log_timestamp("Experiment Start")
```

Explanation:

1. The `start_experiment` function is responsible for handling the actions when the user clicks the "Experiment Start" button.

2. It begins by declaring global variables for the various buttons that are involved in the experiment logging process (`experiment_start_button`, `experiment_end_button`, `interval_start_button`, `task_start_button`, `event_start_button`).

3. The `experiment_start_button` is disabled to prevent the user from clicking it multiple times and starting multiple instances of the experiment.

4. The buttons for logging experiment end, interval start, task start, and event start are enabled (`experiment_end_button.config(state=tk.NORMAL)`, `interval_start_button.config(state=tk.NORMAL)`, `task_start_button.config(state=tk.NORMAL)`, `event_start_button.config(state=tk.NORMAL)`). This allows the user to log these events during the experiment.

5. Finally, the `log_timestamp("Experiment Start")` function is called to log the timestamp for the experiment start. This function is likely defined elsewhere and handles the actual logging of the timestamp with the provided event name ("Experiment Start").

In summary, the `start_experiment` function manages the GUI interactions by enabling and disabling the appropriate buttons and triggers the logging of the experiment start timestamp using the `log_timestamp` function.

```
def end_experiment():
    # Declare global variables for the buttons
    global experiment_start_button, experiment_end_button, interval_start_butt

    # Disable the buttons for logging experiment end, interval start, interval
    experiment_end_button.config(state=tk.DISABLED)
    interval_start_button.config(state=tk.DISABLED)
    interval_end_button.config(state=tk.DISABLED)
    task_start_button.config(state=tk.DISABLED)
    task_end_button.config(state=tk.DISABLED)

    # Enable the "Experiment Start" button
    experiment_start_button.config(state=tk.NORMAL)

    # Log the timestamp for the experiment end
    log_timestamp("Experiment End")
```

Explanation:

1. The `end_experiment` function is responsible for handling the actions when the user clicks the "Experiment End" button.

2. It begins by declaring global variables for the various buttons involved in the experiment logging process (`experiment_start_button`, `experiment_end_button`, `interval_start_button`, `task_start_button`, `event_start_button`).

3. The buttons for logging experiment end, interval start, interval end, task start, and task end are disabled to prevent the user from logging these events after the experiment has ended (`experiment_end_button.config(state=tk.DISABLED)`,
`interval_start_button.config(state=tk.DISABLED)`,
`interval_end_button.config(state=tk.DISABLED)`,
`task_start_button.config(state=tk.DISABLED)`,
`task_end_button.config(state=tk.DISABLED)`).

4.The"ExperimentStart"button is enabled (`experiment_start_button.config(state=tk.NORMAL)`) to allow the user to start a new experiment once the current one has ended.

5. Finally, the `log_timestamp("Experiment End")` function is called to log the timestamp for the experiment end. This function is likely defined elsewhere and handles the actual logging of the timestamp with the provided event name ("Experiment End").

In summary, the `end_experiment` function manages the GUI interactions by enabling and disabling the appropriate buttons and triggers the logging of the experiment end timestamp using the `log_timestamp` function.

start_task function:

```
def start_task():
    # Disable the "Task Start" button
    task_start_button.config(state=tk.DISABLED)

    # Enable the "Task End" button
    task_end_button.config(state=tk.NORMAL)

    # Log the timestamp for the task start
    log_timestamp("Task Start")
```

Explanation:

- The **start_task** function is called when the user clicks the "Task Start" button.
- It disables the "Task Start" button to prevent multiple starts of the same task.
- The "Task End" button is enabled to allow the user to end the task.
- The **log_timestamp** function is called with the event name "Task Start" to log the timestamp for the task start

end_task function:

```
def end_task():
    # Disable the "Task End" button
    task_end_button.config(state=tk.DISABLED)

    # Enable the "Interval Start" button
    interval_start_button.config(state=tk.NORMAL)

    # Log the timestamp for the task end
    log_timestamp("Task End")
```

Explanation:

- The **end_task** function is triggered when the user clicks the "Task End" button.
- It disables the "Task End" button to prevent multiple ends of the same task.
- The "Interval Start" button is enabled to allow the user to start the next interval.
- The **log_timestamp** function is called with the event name "Task End" to log the timestamp for the task end.

start_interval function:

```
def start_interval():
    # Disable the "Interval Start" button
    interval_start_button.config(state=tk.DISABLED)

    # Enable the "Interval End" button
    interval_end_button.config(state=tk.NORMAL)

    # Log the timestamp for the interval start
    log_timestamp("Interval Start")
```

Explanation:

- The **start_interval** function is executed when the user clicks the "Interval Start" button.
- It disables the "Interval Start" button to prevent multiple starts of the same interval.
- The "Interval End" button is enabled to allow the user to end the interval.
- The **log_timestamp** function is called with the event name "Interval Start" to log the timestamp for the interval start

end_interval function:

```
def end_interval():
    # Disable the "Interval End" button
    interval_end_button.config(state=tk.DISABLED)

    # Enable the "Task Start" button
    task_start_button.config(state=tk.NORMAL)

    # Log the timestamp for the interval end
    log_timestamp("Interval End")
```

Explanation:

- The **end_interval** function is triggered when the user clicks the "Interval End" button.
- It disables the "Interval End" button to prevent multiple ends of the same interval.
- The "Task Start" button is enabled to allow the user to start the next task.
- The **log_timestamp** function is called with the event name "Interval End" to log the timestamp for the interval end.

The **start_event** function is a simple function that logs the timestamp for the start of an event. Here's the code snippet:

```
def start_event():
    log_timestamp("Event Start")
```

Explanation:

- When the **start_event** function is called, it invokes the **log_timestamp** function and passes the event name "Event Start" as an argument.
- The **log_timestamp** function is responsible for logging the timestamp of the event, including the event name and the corresponding GMT timestamp in the Finnish timezone.

These functions handle the button states and invoke the **log_timestamp** function to log the corresponding timestamps for different events (task start, task end, interval start, interval end, start event).

```
def save_logged_info():
    filename = folder_path + "/" + participant_name + ".txt"
    with open(filename, "w") as file:
        file.write("Experiment Name: {}\n".format(experiment_name))
        file.write("Participant Name: {}\n".format(participant_name))
        file.write("Timestamp\t\t\tEvent\n")
        for timestamp in timestamps:
            file.write("{}\t{}\n".format(timestamp[1], timestamp[0]))

    print("Logged information saved to: {}".format(filename))
    text.insert(END, f"\nLogged information saved to: {filename}\n")

    experiment_name_entry.delete(0, END)
    participant_name_entry.delete(0, END)
    Folder_path_entry.delete(0, END)
    # Enable the experiment name and participant name entry fields
    experiment_name_entry.config(state=tk.NORMAL)
    participant_name_entry.config(state=tk.NORMAL)
```

Explanation:

- The function begins by constructing the filename for the text file based on the folder path and participant name.
- It then opens the file in write mode using a **with** statement, which ensures that the file is properly closed after writing.
- The function writes the experiment name, participant name, and a header line ("Timestamp\t\t\tEvent") to the file.

- It iterates over the **timestamps** list, which contains tuples of event names and GMT timestamps, and writes each timestamp and event name to a new line in the file.
- After writing the logged information, it prints a message indicating the file where the information is saved and inserts the same message into the text widget for display.
- Lastly, the function performs some cleanup tasks by clearing the input fields (experiment name, participant name, folder path) and re-enabling the disabled entry fields and folder button for further use.

This function provides the functionality to save the logged information in a structured manner, making it easier to analyze and review the recorded events and timestamps.

```
def record_audio(file_path_1):
    CHUNK = 1024
    FORMAT = pyaudio.paInt16
    CHANNELS = 1
    RATE = 44100

    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT,
                    channels=CHANNELS,
                    rate=RATE,
                    input=True,
                    frames_per_buffer=CHUNK)

    frames = []
    start_time = time.time()

    while not stop_event.is_set():
        data = stream.read(CHUNK)
        frames.append(data)
```

```
stream.stop_stream()
stream.close()
p.terminate()

filename = f"audio_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}.w
audio_file_path = os.path.join(file_path_1, filename)

wf = wave.open(audio_file_path, 'wb')
wf.setnchannels(CHANNELS)
wf.setsampwidth(p.get_sample_size(FORMAT))
wf.setframerate(RATE)
wf.writeframes(b''.join(frames))
wf.close()
```

Explanation:

- The function begins by defining constants such as **CHUNK** (the number of audio frames per buffer), **FORMAT** (the audio data format), **CHANNELS** (the number of audio channels), and **RATE** (the sampling rate).
- It creates a PyAudio instance **p** and opens a stream **stream** for audio input using the specified format, channels, rate, and buffer size.
- An empty list **frames** is created to store the audio frames captured from the microphone.
- The variable **start_time** is set to the current time using **time.time()**.
- Inside the **while** loop, the function continuously reads audio data from the stream in chunks of size **CHUNK** and appends them to the **frames** list.
- The loop continues until the **stop_event** flag is set (presumably by another part of the code not shown here).
- Once the loop exits, the stream is stopped, closed, and the PyAudio instance is terminated.

- A filename is generated based on the current timestamp, and the audio file path is constructed by joining **file_path_1** (the directory path) and the filename.
- A new wave file is created using **wave.open** in write mode ('wb').
- The audio file's parameters such as the number of channels, sample width, and frame rate are set using the corresponding methods of the wave file object **wf**.
- The audio frames in **frames** are joined together into a single byte string using **b''.join(frames)**, and then written to the wave file using **wf.writeframes**.
- Finally, the wave file is closed with **wf.close ()**.

This function allows for the recording of audio from the microphone and saves it as a wave file in the specified directory. The audio capture continues until the **stop_event** is set.

```
def mouse_keyboard_stream(file_path_1):
    global keyboard_listener, mouse_listener

    timestamp = int(time.time())

    # File paths for keyboard and mouse recordings
    keyboard_file = os.path.join(file_path_1, f'keyboard_data_{timestamp}.
    mouse_file = os.path.join(file_path_1, f'mouse_data_{timestamp}.txt')

    # Open the text files for writing
    keyboard_file_handle = open(keyboard_file, 'w')
    mouse_file_handle = open(mouse_file, 'w')
```

Explanation:

- The function begins by defining two global variables **keyboard_listener** and **mouse_listener**. These variables likely represent listeners or objects that capture keyboard and mouse events.

- The variable **timestamp** is assigned the current time in seconds since the epoch using **int(time.time())**. This timestamp is used to generate unique filenames for the keyboard and mouse data files.
- The file paths for the keyboard and mouse recordings are created by joining the **file_path_1** (the directory path) and the respective filename generated using the timestamp.
- Two file handles, **keyboard_file_handle** and **mouse_file_handle**, are created by opening the keyboard and mouse data files in write mode ('w').

This function sets up the necessary variables and file handles to capture and record keyboard and mouse data. The specifics of how the keyboard and mouse data are captured and written to the files are not shown in the provided code snippet.

```
def on_keyboard_press(key):
    # Log the key and timestamp
    log_data = f"Key pressed: {key} at timestamp {time.time()}\n"
    print(log_data)

    # Write the data to the keyboard file
    keyboard_file_handle.write(log_data)
    keyboard_file_handle.flush()
```

Explanation:

- The function takes a **key** parameter, which represents the key that was pressed on the keyboard.
- A log message is created using an f-string, containing the information about the pressed key and the current timestamp.
- The log message is printed to the console using **print(log_data)**.

- The log message is then written to the keyboard file using the **write** method of the **keyboard_file_handle** file handle.
- After writing the data, the **flush** method is called on the file handle to ensure that the data is immediately written to the file.

This function captures the pressed key and the timestamp, prints the information to the console, and writes it to the keyboard file for recording the keyboard input data.

The**on_keyboard_release** function, starts listening for keyboard events using the **keyboard.Listener**, and creates a LabStreamingLayer (LSL) outlet for mouse data. Here's an explanation of the code:

```
def on_keyboard_release(key):
    # Handle key release events if needed
    pass
```

The **on_keyboard_release** function is a callback function that is triggered when a key is released on the keyboard. In this case, the function does not contain any specific implementation and simply passes.

```
keyboard_listener = keyboard.Listener(on_press=on_keyboard_press, on_release=
keyboard_listener.start()
```

This code sets up a **keyboard.Listener** instance to listen for keyboard events. The **on_press** parameter is set to the previously defined **on_keyboard_press** function, which will be called when a key is pressed. The **on_release** parameter is set to the **on_keyboard_release** function, which will be called when a key is released. Finally, the **keyboard_listener.start()** method is called to start listening for keyboard events.

```
info = StreamInfo('MouseData', 'Mouse', 2, 0, 'float32', 'mouse-id')
outlet = StreamOutlet(info)
```

In this part, an LSL outlet for mouse data is created. The **StreamInfo** class is used to define the properties of the stream. Here, the stream is named '**MouseData**' with a type of '**Mouse**'. It has 2 channels, which typically represent the x and y coordinates of mouse movements. The sample rate is set to 0, indicating that the stream is event-based rather than continuous. The data format is specified as '**float32**', representing single-precision floating-point values. The '**mouse-id**' serves as the unique identifier for the stream.

Finally, an outlet is created using the **StreamOutlet** class and the defined **info**. This outlet allows you to send mouse data through the LSL stream, making it available for logging or further processing by other applications.

By combining these components, you can listen for keyboard events and create an LSL outlet for mouse data, enabling you to capture and record both keyboard and mouse activities for your experiment or application.

```
def on_mouse_click(x, y, button, pressed):
    if pressed:
        # Log the mouse click coordinates and timestamp
        log_data = f"Mouse click at ({x}, {y}) at timestamp {time.time()}\n"
        print(log_data)

        # Write the data to the mouse file
        mouse_file_handle.write(log_data)
        mouse_file_handle.flush()

        # Send the mouse data through LSL
        outlet.push_sample([x, y], time.time())
```

The **on_mouse_click** function is a callback function that is triggered when a mouse button is clicked. It takes the parameters **x** and **y** representing the coordinates of the click, **button** representing the button that was clicked, and **pressed** indicating whether the button was pressed or released. In this implementation, it checks if the button was pressed, and if so, it logs the mouse

click coordinates and timestamp, writes the data to the mouse file, and sends the mouse data through LSL using the **outlet** created earlier.

```
mouse_listener = mouse.Listener(on_click=on_mouse_click)
mouse_listener.start()
```

This code sets up a **mouse.Listener** instance to listen for mouse events. The **on_click** parameter is set to the **on_mouse_click** function, which will be called when a mouse button is clicked. The **mouse_listener.start()** method is then called to start listening for mouse events.

```
while not stop_event.is_set():
    time.sleep(0.1)
```

This loop keeps the program running until the **stop_event** is set. It uses a small sleep duration of 0.1 seconds to avoid excessive CPU usage.

```
# Stop the keyboard and mouse listeners
keyboard_listener.stop()
mouse_listener.stop()

# Close the file handles
keyboard_file_handle.close()
mouse_file_handle.close()

print("Recording stopped. Data saved to", file_path_1)
reset_interface()
```

These lines of code are executed when the recording is stopped. The keyboard and mouse listeners are stopped using their respective **stop()** methods. The file handles for the keyboard and mouse files are closed. A message is printed indicating that the recording has stopped and the data is saved to the specified **file_path_1**. Finally, the **reset_interface()** function is called, which could be a function to reset the user interface or perform any necessary cleanup tasks.

This code snippet completes the implementation of recording keyboard and mouse data, saving it to files, and streaming the mouse data through LSL.

Here's an explanation of the **start_recording** function

```
def start_recording():
    global file_path_1, audio_thread, mouse_keyboard_thread

    file_path_1 = Folder_path_entry.get()

    if not os.path.isdir(file_path_1):
        return
```

In this snippet, the function begins by retrieving the value entered in the Folder_path_entry widget and assigning it to the file_path_1 variable. It then checks if the specified file_path_1 represents a valid directory. If it is not a directory, the function returns and the recording process is not initiated.

```
if audio_var.get() == 1:
    audio_thread = threading.Thread(target=record_audio, args=(file_path_1,))
    audio_thread.start()
```

This snippet checks if the **audio_var** variable is set to 1, indicating that the audio recording option is selected. If it is selected, a new thread (**audio_thread**) is created. The **record_audio** function is set as the target of the thread, and the **file_path_1** is passed as an argument. The thread is then started to initiate the audio recording process.

```
if keyboard_var.get() == 1 or mouse_var.get() == 1:
    mouse_keyboard_thread = threading.Thread(target=mouse_keyboard_stream, ar
    mouse_keyboard_thread.start()
```

This snippet checks if either the **keyboard_var** or **mouse_var** variables are set to 1, indicating that the keyboard or mouse recording option is selected. If either option is selected, a new thread (**mouse_keyboard_thread**) is created. The **mouse_keyboard_stream** function is set as the target of the thread, and the **file_path_1** is passed as an argument. The thread is then started to initiate the keyboard and mouse recording process.

```
start_button.config(state=DISABLED)
```

This snippet disables the "Start Recording" button by setting its state to **DISABLED**. This prevents multiple recordings from being started simultaneous.

In summary, the **start_recording** function retrieves the file path for the recording, checks if it is a valid directory, and initiates separate threads for audio recording and keyboard/mouse recording if the corresponding options are selected. It also disables the "Start Recording" button and starts the periodic check of the recording status.

```
def stop_recording():
    global stop_event

    stop_event.set()
```

In this snippet, the **stop_recording** function sets the **stop_event** variable to indicate that the recording should be stopped. This variable is typically used as a flag to control the recording loops in the audio and mouse/keyboard threads.

```
def check_recording_status():
    if audio_thread and audio_thread.is_alive():
        window.after(100, check_recording_status)  # Check again after 100
    elif mouse_keyboard_thread and mouse_keyboard_thread.is_alive():
        window.after(100, check_recording_status)  # Check again after 100
```

This snippet defines the **check_recording_status** function, which is called periodically to monitor the status of the recording threads. It checks if the **audio_thread** is alive (indicating that the audio

recording is still in progress) or if the **mouse_keyboard_thread** is alive (indicating that the keyboard or mouse recording is still in progress). If either thread is alive, the function schedules itself to be called again after 100 milliseconds using the **window.after** method.

```
def reset_interface():
    global keyboard_listener, mouse_listener, stop_event

    if keyboard_listener:
        keyboard_listener.stop()

    if mouse_listener:
        mouse_listener.stop()

    stop_event.clear()
```

This snippet defines the **reset_interface** function, which is responsible for resetting the recording interface and cleaning up resources. It stops the keyboard and mouse listeners (**keyboard_listener** and **mouse_listener**) if they exist, and clears the **stop_event** variable to ensure that the recording is completely stopped. This function is typically called after the recording is finished.

In summary, the **stop_recording** function sets the stop event to stop the recording process, the **check_recording_status** function periodically checks the status of the recording threads, and the **reset_interface** function cleans up the recording interface and resources. These functions work together to provide control and monitoring of the recording process.