Timer-Based Triggers for Scheduling Events: A Framework for Day-to-Day, Week-to-Week, Month- to-Month, and Annual Automation

Omkar Wagle[1], Anand Kumar Singh[2]

Abstract

This paper explores a robust SysTimer[1] based scheduling framework designed to automate tasks based on predefined intervals, including Single-Event, Day-to-Day, Week-to-Week, Month-to-Month, and Annual scheduling. The framework efficiently manages repetitive and time-sensitive activities, ensuring flexibility, scalability, and accuracy across diverse applications. Each scheduling type is defined with its use cases, implementation guidelines, and validation logic, minimizing human error and enhancing reliability. Flowcharts are presented to illustrate the complete lifecycle of event creation, validation, SysTimer[1] execution, and re-scheduling. The proposed framework demonstrates practical value in various scenarios, such as automating periodic maintenance tasks in unattended environments, thereby maintaining optimal conditions and reducing manual effort.

Keywords: SysTimer, MQTT, SDK(Software Development Kit)

1. Introduction

In the era of smart automation and IoT, managing and executing scheduled tasks across various devices efficiently is crucial. This paper presents a robust, SysTimer[1] based scheduling automation framework that addresses these challenges by integrating three key components for seamless operation and control. The first component is a mobile application (Android/iOS) that enables users to create automated tasks with scheduling options such as Single-Event, Day-to-Day, Week-to-Week, Month-to-Month, and Annual. The second component is the gateway (SOC) interface, which validates incoming events, discards invalid ones with error notifications, and stores valid events in a SQLite[2] database with an assigned scheduling type variable. Once the timer expires, the SysTimer[1] mechanism ensures precise execution of scheduled tasks by sending a command to the application controlling Zigbee/Zwave device through MQTT[3]. The third component consists of Zigbee/Zwave devices, where actions are performed. This integrated framework offers an efficient, reliable, and scalable solution for automated task scheduling and execution, applicable to various domains such as home automation, facility management, and industrial control systems.

2. Scheduling Types for Automated Triggers

2.1.Single-Event Trigger

Executes only once in its lifetime. A Single-Event Trigger is designed to execute only once at a predefined time, making it ideal for handling unique events or tasks that do not require repetition. Once the action is completed, the trigger does not repeat unless it is manually set again. This type of trigger is commonly used for scheduling important events like software updates, bill payments, or Single-Event notifications. By automating these singular tasks, it ensures timely execution while minimizing the risk of human error or oversight.

2.2.Day-to-Day Trigger

Executes once every 24 hours. A Day-to-Day Trigger is designed to execute an action at the same time every day, ensuring consistency and reliability in repetitive tasks. Once set, it functions automatically without requiring any manual intervention, making it ideal for routine processes such as system backups, automated reports, or scheduled device operations. By eliminating the need for constant monitoring, Day-to-Day Trigger help streamline workflows, improve efficiency, and reduce the risk of human error in time-sensitive activities Used for repetitive Day-to-Day tasks.

2.3.Week-to-Week Trigger

Executes once every 7 days. A Week-to-Week Trigger is programmed to execute on specific days of the week at a predetermined time, ensuring that recurring tasks are carried out consistently. It is particularly useful for scheduled Week-to-Week activities such as maintenance routines, team meetings, or automated system updates. By automating these recurring tasks, Week-to-Week Trigger help maintain organization, improve efficiency, and reduce the need for manual oversight, allowing users to focus on other priorities while ensuring essential operations run smoothly.

2.4.Month-to-Month Trigger

If the date is valid, the trigger will occur every month on the same date. A Month-to-Month is designed to execute on a specific date each month, ensuring that periodic tasks are carried out consistently without manual intervention. A Month-to-Month Trigger is particularly useful for managing recurring responsibilities such as billing cycles, subscription renewals, salary processing, and maintenance schedules. By automating these Month-to-Month tasks, it helps streamline operations, improve efficiency, and reduce the risk of missed deadlines, making it an essential tool for both personal and business management.

2.5.Annual Trigger

Executes on the same date every year, considering leap years. An Annual Trigger is designed to run once a year on a fixed date, making it ideal for managing annual events and long-term planning. Annual Trigger ensures that important tasks, such as tax filing deadlines, performance reviews, policy renewals, and special occasions like birthdays and anniversaries, are never forgotten. By automating these Annual reminders and actions, this event helps maintain organization, reduces the risk of oversight, and allows individuals and businesses to plan efficiently for recurring obligations and celebrations.

3. Trigger Configuration Attributes

3.1.Trigger Identifier (trigger_id)

A unique identifier assigned to each event execution, and also the primary key in the SQLite[2] database where the event data is stored. This ensures that each event execution record can be uniquely identified and efficiently retrieved from the SQLite[2] database. As the primary key in the SQLite[2] database, the trigger_id serves as the main reference point for each event execution record. It ensures that each event execution is stored and retrieved independently. The trigger_id links all associated data—such as event type, command payload, timestamps, etc.- to a specific event execution instance, making it essential for SQLite[2] database queries, indexing, and data management.

3.2.Trigger Execution Timestamp (trigger_exec_tm)

The Trigger Execution Timestamp is an essential attribute that defines the exact time when an event is

scheduled to execute. This timestamp represents the scheduled timestamp at which the event should trigger its execution, regardless of whether it has been triggered previously. It serves as a reference point for the system to calculate the execution timestamp for recurring triggers, or to track when a Single-Event trigger should run**.** The Trigger Execution Timestamp is the reference point used by the system to calculate the next execution timestamp. This timestamp determines when the event should be triggered.

This attribute is typically stored as a Unix timestamp (in milliseconds since January 1, 1970, UTC). Example: 1741599600000 (for Mon Mar 10, 2025, 09:40:00 GMT + 0000 in Unix time format).

3.3.Last Trigger Execution Timestamp (trigger_last_exec_tm)

The trigger_last_exec_tm is a critical parameter used to track when an event was last executed or updated. It represents the timestamp (in milliseconds) of the last time the event was triggered or executed. This timestamp helps in scheduling the next execution of the event, especially for recurring trigger types like Day-to-Day, Week-to-Week, Month-to-Month, and Annual.

This attribute is typically stored as a Unix timestamp (in milliseconds since January 1, 1970, UTC).

3.4.Trigger Type (trigger_type)

The trigger_type parameter specifies the type or frequency of the event's execution. This variable helps differentiate between different types of triggers based on how often or when they should be executed. The values are typically represented as an integer corresponding to different validation mechanisms accompany each trigger type to ensure proper handling of edge cases, such as invalid dates for all five types of triggers. For Month-to-Month Trigger, it also checks the valid date in a month and the next possible execution in the next month, and for Annual events, it focuses on leap year validation.

3.5.Time in milliseconds (time_in_ms)

The time_in_ms parameter holds the time (in milliseconds) that must pass before the event is triggered and executed. This delay is based on the type of trigger (such as Single-Event, Day-to-Day, Week-to-Week, Month-to- Month, and Annual), the current timestamp, and any other conditions that affecting the execution timing. This parameter is used to calculate the event’s execution timestamp. By storing the time in milliseconds, it provides precision for executing events after specific delays.

3.5.1. Single-Event: trigger_type = 0

This event is designed to execute only once. Once the event has been executed for the first time, it will not be executed again.

Upcoming Execution:

There are two possible conditions to consider when calculating the time until the event should execute:

Condition 1: When trigger_exec_tm is greater than the current time (current_tm):

If the scheduled execution time (trigger_exec_tm) is still in the future (i.e., the event has not yet triggered), the delay or time until the next execution (denoted as time_in_ms) can be calculated by finding the difference between the scheduled execution time and the current time. The formula for this is:

$$time\_in\_ms = trigger\_exec\_tm - current\_tm$$

Condition 2: When trigger_exec_tm is less than the current time (current_tm):

If the scheduled execution time (trigger_exec_tm) is in the past (i.e., the event was supposed to trigger earlier but hasn't), it means the event has already expired, and there will be no further execution of the event.

Next Execution:

Since this is a Single-Event with a one-time execution time, once the event has been executed or missed, there will be no next execution. This event is designed to trigger only once, and after that, it will not trigger again.

Where:

- trigger_exec_tm: The timestamp that indicates when the Single-Event is scheduled to execute.
- current_tm: The timestamp of the current time when the calculation is being performed.

3.5.2. Day-to-Day: trigger_type = 1

For Day-to-Day events, the system checks the relationship between the event execution time (trigger_exec_tm) and the current time (current_tm) to determine if the event needs to be executed and when the next execution will happen. There are following conditions to consider:

Upcoming Execution:

Condition 1: When trigger_exec_tm is greater than current_tm:

If the scheduled execution time (trigger_exec_tm) for the event is in the future (i.e., the event has not been triggered yet), the delay, or the time in milliseconds until the event's execution can be calculated by finding the difference between the scheduled execution time and the current time. The formula for this is:

$$time\_in\_ms = trigger\_exec\_tm - current\_tm$$

Condition 2: When trigger_exec_tm is less than current_tm**:**

If the scheduled execution time (trigger_exec_tm) is in the past (i.e., the event should have already been triggered), it means the event missed its execution. In this case, the next execution time will be set to the same time on the next day (24 hours later). The delay (time in milliseconds) is calculated as:

$$time\_in\_ms = (trigger\_exec\_tm + (24 * 60 * 60 * 1000)) - current\_tm$$

Once the event is executed, trigger_last_exec_tm is updated to the current time (current_tm), indicating the last execution time:

$$trigger\_last\_exec\_tm = current\_tm$$

Next Execution:

The next execution time for the event will always be the same time on the next day. This can be calculated by adding 24 hours (expressed in milliseconds) to the trigger_last_exec_tm:

$$trigger\_exec\_tm = trigger\_last\_exec\_tm + (24 * 60 * 60 * 1000)$$

Where:

- trigger_exec_tm: The next execution time for Day-to-Day Trigger.
- current_tm: The current time refers to the variable or value that stores the date and time at the moment of execution, representing the current system time.
- trigger_last_exec_tm: This parameter is updated after each execution, providing a record of the most

recent execution time of the event. It helps track when the event occurred, allowing for time-based comparisons or scheduling decisions.

- (24 * 60 * 60 * 1000): The number of milliseconds in one day (24 hours).

3.5.3. Week-to-Week: trigger_type = 2

This event type executes Week-to-Week. Based on whether the event execution time (trigger_exec_tm**)** is in the future or has passed, the system calculates the delay (time_in_ms) and schedules the next execution. For Week- to-Week events, the system checks the relationship between the event execution time (trigger_exec_tm) and
the current time (current_tm) to determine if the event needs to be executed and when the next execution will happen. There are following conditions to consider:

Upcoming Execution:

Condition 1: When trigger_exec_tm is greater than current_tm:

When the event execution time (trigger_exec_tm) is in the future (i.e., the event has not yet been executed), the system calculates the delay (time_in_ms) until the scheduled execution as the difference between the event's execution time and the current time. The delay is calculated as:

$$time\_in\_ms = trigger\_exec\_tm - current\_tm$$

Condition 2: When trigger_exec_tm is less than current_tm

When the event execution time (trigger_exec_tm) has already passed (i.e., the event missed its scheduled execution), the system needs to schedule the next execution one week after the missed execution. This ensures the event will execute again after one full week. The delay is calculated by adding 7 days (in milliseconds) to the original execution time (trigger_exec_tm), then subtracting the current time (current_tm). This gives the delay until the next execution. The delay is calculated as:

$$time\_in\_ms = (trigger\_exec\_tm + (7 * 24 * 60 * 60 * 1000)) - current\_tm$$

After the event execution (whether it is executed now or after the delay), the last execution time (trigger_last_exec_tm) will be updated to the current time (current_tm). This update ensures that the system tracks when the event was last executed and can calculate the next execution time accordingly.

$$trigger\_last\_exec\_tm = current\_tm$$

Next Execution:

Once the event is executed, the next execution time is scheduled for one week later, based on the most recent execution. The time delay until the next execution is calculated as:

$$time\_in\_ms = (trigger\_last\_exec\_tm + (7 * 24 * 60 * 60 * 1000)) - current\_tm$$

Where:

- trigger_exec_tm: The next execution time for Week-to-Week Trigger.
- current_tm: The current time refers to the variable or value that stores the date and time at the moment of execution, representing the current system time.
- trigger_last_exec_tm: This parameter is updated after each execution, providing a record of the most recent execution time of the event. It helps track when the event occurred, allowing for time-based comparisons or scheduling decisions.
- (7 * 24 * 60 * 60 * 1000): The number of milliseconds in 7 days.

3.5.4. Month-to-Month: trigger_type = 3

The Month-to-Month event is scheduled to execute once every month. The event's execution time and the current time determine the delay (time_in_ms) before the next execution.

Upcoming Execution:

Condition 1: When trigger_exec_tm is greater than current_tm:

When the event execution time (trigger_exec_tm) is still in the future (i.e., the event hasn't yet been executed), the delay before the next execution is calculated. The time delay is calculated by considering the difference in timestamp between the trigger_exec_tm and the current_tm

$$time\_in\_ms = trigger\_exec\_tm - current\_tm$$

Condition 2: When trigger_exec_tm is less than current_tm

When the event execution time (trigger_exec_tm) has already passed (i.e., the event missed its scheduled execution) Following are the steps to calculate time_in_ms

Case I. If trigger_exec_tm.day > current_tm.day

If the scheduled day (trigger_exec_tm.day) is greater than the current day (current_tm.day), it means the event is still valid in the current month. In this case,

Step 1. Set trigger_exec_tm.month = current_tm.month and exec_tm.year = current_tm.year.
Step 2. We then assign calculated trigger execution time to time_in_ms as month and year are already adjusted.

$$time\_in\_ms = exec\_tm$$

Case II. If trigger_exec_tm.day < current_tm.day

If the scheduled day (exec_tm.day) is less than or equal to the current day (current_tm.day), the event needs to be moved to the next month. We increase exec_tm.month by 1.

Handling Month Overflow:

If trigger_exec_tm.month is greater than 12: This condition handles the case where the month is moved beyond December (i.e., the next month becomes January). In this case,

Step 1. Set trigger_exec_tm.year = exec_tm.year + 1 and
Step 2. trigger_exec_tm.month = 1 (i.e., the next event will be in January of the following year).

Adjusting the Day:

Now that the month and year are adjusted, we check the day of trigger_ exec_tm:

Scenario 1: trigger_exec_tm.day <= 28

If exec_tm.day is 28 or less, no adjustments are needed because every month has at least 28 days. In this case, exec_tm.day will remain the same as the previous month, and the event is scheduled accordingly.

$$time\_in\_ms = trigger\_exec\_tm \text{ (the same day in the next month)}$$

Scenario 2: trigger_exec_tm.day > 28

If the scheduled day (exec_tm.day) is greater than 28, we need to consider the number of days in the next month:

Case 2.1: February (trigger_exec_tm.month = 2)

If the event is scheduled for February, we check if the year is a leap year or not, If the year is a leap year, February will have 29 days, so the event will be scheduled for February 29th. If the year is not a leap year, February will have only 28 days, so the event will be scheduled for February 28th.

Case 2.2: April, June, September, November (trigger_exec_tm.month = 4, 6, 9, 11)

These months have only 30 days. If the scheduled day is greater than 30, we adjust it to April 30, June 30, September 30, or November 30, depending on the month.

Case 2.3: January, March, May, July, August, October, December (exec_tm.month = 1, 3, 5, 7, 8, 10, 12)

These months have 31 days.If the scheduled day is greater than 31, we adjust it to 31 (the last day of the month).

Next Execution

After the event is executed, the system calculates the time delay (time_in_ms) for the next execution**.**
If the day of the last execution (i.e., trigger_last_exec_tm) is valid in the next month (i.e., it's a valid day for the upcoming month):

Δt = *(difference between last exec day and next exec day) * 24 * 60 * 60 * 1000*

If the day of the last execution is not valid in the next month (for example, 31st of April which is not valid):

$time\_in\_ms = trigger\_last\_exec\_tm + \Delta t$

Where:

- trigger_exec_tm: The next execution time for Month-to-Month Trigger.
- current_tm: The current time refers to the variable or value that stores the date and time at the moment of execution, representing the current system time.
- trigger_last_exec_tm: This parameter is updated after each execution, providing a record of the most recent execution time of the event. It helps track when the event occurred, allowing for time-based comparisons or scheduling decisions.
- Δt**:** Δt is the time delay (in milliseconds) before the event gets triggered again. It's the amount of time that needs to pass from the last execution time to the next execution time.

This method ensures that the event is scheduled to execute on the **same day** of each month, but if that day doesn't exist in the next month (for example, 31st January), the event will be scheduled on the last valid day of the next month.

3.5.5. Annual: trigger_type = 4

For Annual events, the upcoming execution is calculated by considering the difference between the current time and the scheduled execution time, based on whether the scheduled execution is in the future or the past. There are two main conditions for determining the delay until the next execution.

Upcoming Execution

Condition 1: When trigger_exec_tm is greater than current_tm:

If the trigger_exec_tm is greater than current_tm (i.e., the event is scheduled to execute in the future). The time delay is calculated by considering the difference in timestamp between the trigger_exec_tm and the current_tm

$time\_in\_ms = trigger\_exec\_tm - current\_tm$

Condition 2. When trigger_exec_tm is less than current_tm

If the event execution time has already passed, it indicates that the event was supposed to be executed earlier in the current year. In this case, the next execution will occur in the following year. If the scheduled execution time's year (trigger_exec_tm.year) is less than the current year (current_tm.year), we need to determine whether to update the year or not. First, we check if the scheduled month (trigger_exec_tm.month) is greater than the current month (current_tm.month). If it is, this means the event can still occur within the current year, so we update the execution time's year to match the current year, keeping the same month and day, and calculate the time in milliseconds (time_in_ms) based on this updated date. However, if the scheduled month is less than the current month, it means that the intended execution month has already passed in the current year. In this case, we increment the scheduled execution year by one to ensure the event is scheduled for the following year. We then calculate the time in milliseconds for this updated date. Additionally, when considering leap years, if the scheduled month and day are in February and the day is set to the 29th, we must ensure that the updated year is a leap year (divisible by 4 and not divisible by 100, unless also divisible by 400). If the year after incrementing is not a leap year and the day is set to the 29th of February, we must adjust the day to the 28th to maintain a valid date. This way, the logic ensures that the event is correctly scheduled while accounting for both past and future scenarios, as well as leap year considerations.

$$trigger\_exec\_tm.year = \begin{cases} current\_tm.year, & if\ trigger\_exec\_tm.month > current\_tm.month \\ current\_tm.year + 1, & if\ trigger\_exec\_tm.month < current\_tm.month \end{cases}$$

Leap Year Adjustment (if February 29): The given logic handles cases where the scheduled date is February 29 in a non-leap year. Since non-leap years do not have February 29, the logic updates the date to February 28 to ensure validity. The delay is calculated as,

Leap Year Check:

$$time\_in\_ms = trigger\_exec\_tm - current\_tm$$

$$is\_leap\_year(year) = \begin{cases} True, & if\ (year\%4 = 0)\ and\ ((year\%100 \neq 0)\ or\ (year\%400 = 0) \\ False, & Otherwise \end{cases}$$

Next Execution

The next execution time (next_exec_tm) is determined by adding one year's worth of time (in milliseconds) to the last execution time (trigger_last_exec_tm). The formula to calculate this is**:**

*time_in_ms = trigger_last_exec_tm + ((days_in_year) * 24 * 60 * 60 *1000)*

Where:

- trigger_exec_tm: The next execution time for Annual Trigger.
- current_tm: The current time refers to the variable or value that stores the date and time at the moment of execution, representing the current system time.
- trigger_last_exec_tm: This parameter is updated after each execution, providing a record of the most recent execution time of the event. It helps track when the event occurred, allowing for time-based comparisons or scheduling decisions.
- days_in_year – Number of days from last execution day till next execution day
- (24 * 60 * 60 * 1000): The number of milliseconds in a day.

3.6. *SysTimer ID*

When a new Event is created, the application assigns a unique SysTimer[1] ID, trigger ID, and calculates the time in milliseconds. These configuration details are stored in a linked list during the trigger setup. Once the time in milliseconds reaches 0, the SysTimer[1] retrieves the corresponding ID from the linked list, executes the trigger, and depending on the event's trigger type, determines whether to schedule the event for the next execution or remove it from the linked list.

4. DST Mode

While the framework effectively accounts for leap year adjustments, other potential edge cases, such as daylight-saving time (DST) transitions and time zone differences, must also be considered to ensure precise event scheduling across various regions. These factors can lead to unexpected shifts in scheduled events, especially in global applications where time adjustments occur periodically.

4.1.Handling Daylight Saving Time (DST)

Daylight saving time changes occur in many regions, where clocks are adjusted forward by one hour in spring and backward by one hour in autumn. This can result in:

- Skipped events: If an event is scheduled during the missing hour (e.g., 2:30 AM when clocks jump to 3:00 AM), it may never execute.
- Duplicate executions: If an event is scheduled at a time that repeats when clocks are set back (e.g., 1:30 AM appearing twice), it may trigger twice.

To mitigate these issues, the framework should:

- Store all execution timestamps in Coordinated Universal Time (UTC) and convert to local time dynamically at runtime.
- Implement DST-aware scheduling, ensuring events are adjusted based on the system’s time zone settings. Allow user-defined execution policies, such as skipping or delaying events affected by DST transitions

4.2.Managing Time Zone Differences

For applications operating across multiple time zones, discrepancies in scheduled execution can arise when users move between time zones or schedule tasks remotely. To address this, the framework should:

- Maintain all scheduled events in UTC format, ensuring a consistent reference point.
- Support time zone synchronization mechanisms, where event timestamps are dynamically adjusted based on the device's configured time zone.
- Provide an option for fixed-time execution (e.g., executing at 9:00 AM local time regardless of time zone) versus absolute-time execution (e.g., executing at a specific UTC time globally).

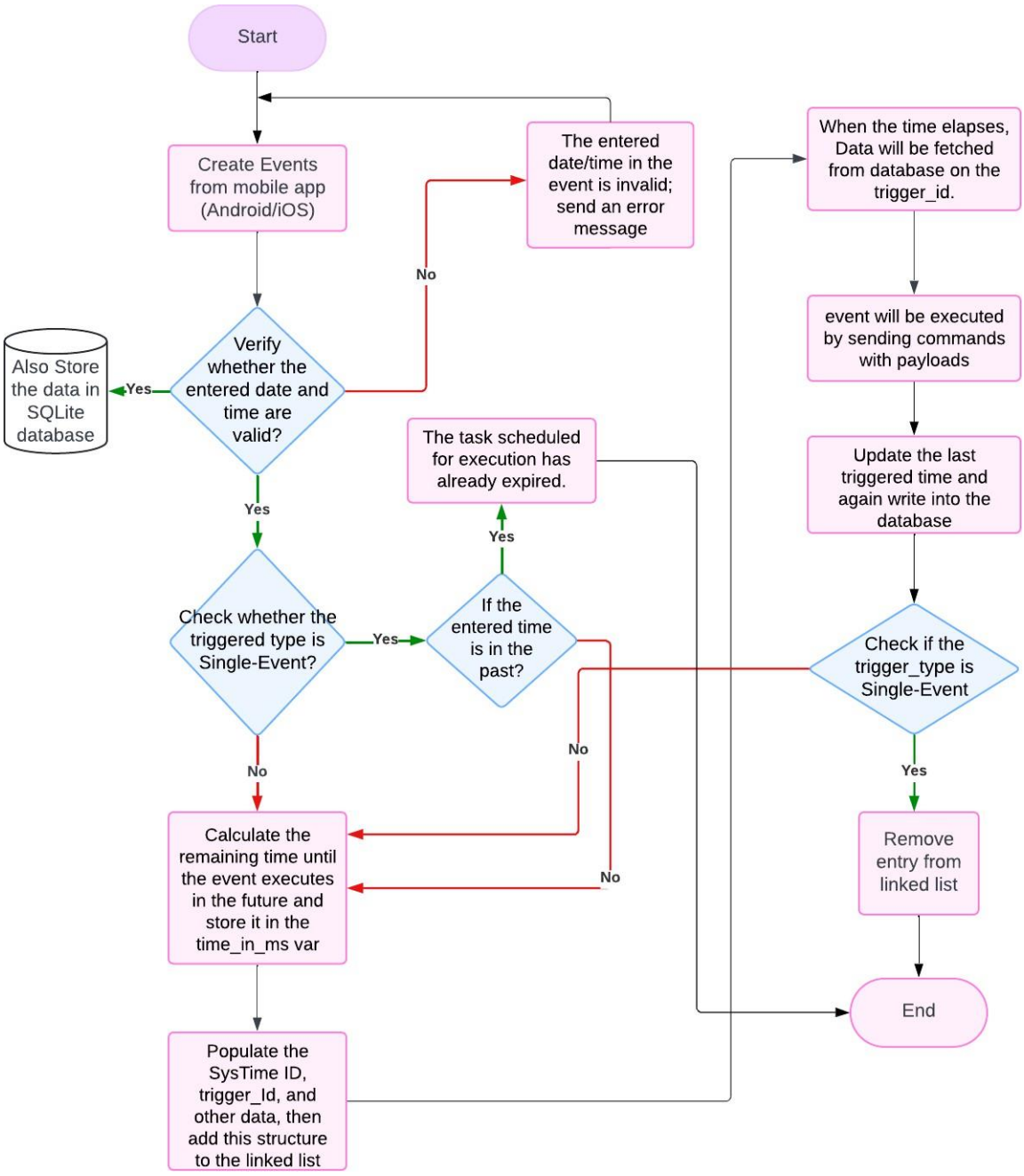
By addressing these time-dependent edge cases, the scheduling framework can ensure greater reliability and accuracy across various regions and time conditions, making it more resilient for real-world deployment in IoT automation, facility management, and industrial control applications.

5. Functioning of Event

The event creation and execution process follow a structured and automated workflow. Below is a detailed explanation of how the events operate:

1. Event Creation: The event is initially created through a mobile application, either on Android or iOS. Users configure the event settings and parameters based on their specific requirements.
2. SDK Reception: Once the event is created, it is sent to the SDK library, which handles further processing.
3. Attribute Validation: The SDK performs a comprehensive validation of each attribute within the event to ensure correctness and consistency. This step verifies that the event meets all the necessary criteria and does not contain errors.
4. Database Storage: After successful validation, the event is written to the database for permanent storage and easy retrieval.
5. Time Calculation: The system calculates the time required to trigger each specific event in milliseconds. This delay is determined based on the trigger type (e.g., Single-Event, Day-to-Day, Week-to-Week, Month-to- Month, and Annual trigger).
6. Linked List Creation: A linked list is created to manage all the events efficiently. This list helps in organizing and maintaining the events systematically.
7. Timer Assignment: Once the time is calculated, each trigger is assigned a unique SysTimer[1] ID. The calculated time (in milliseconds) is associated with this ID to track when the event needs to be executed.
8. Adding Event to Linked List: After all necessary parameters are filled and the SysTimer[1] is assigned, the event is added to the linked list for continuous monitoring and execution.
9. Event Execution: When the time expires for a specific event, the associated command ID and payload linked to that event are retrieved from SQLite[2] database and sent to the designated devices. This ensures that the intended action is performed on time.
10. Updating Execution Status: After the event is successfully executed, the relevant data is fetched from the event table. The system then updates the last execution time in the database to keep track of the event's execution history.
11. Command Transmission: Finally, the command is sent from the SDK layer to the intended device via the MQTT[3] protocol, ensuring real-time communication and efficient execution of the event.

This workflow ensures that events are created, managed, and executed in an organized and automated manner, maintaining the accuracy and timeliness of each operation.



6. Comparison with Existing Scheduling Methods

The SysTimer[1]-based scheduling framework provides a software-driven approach to event scheduling, making it a viable alternative to traditional scheduling methods such as Linux Cron Jobs, RTOS Task Schedulers, and Hardware Timers. Each of these mechanisms has trade-offs in precision, scalability, real-time behavior, resource usage, and persistence, which determine their suitability for specific applications. Below is comparison with some of pro and con with other available methods:

1. Linux cron jobs are widely used for system-level automation but are limited to second-level granularity and lack built-in mechanisms for handling missed executions or leap year adjustments. While they are highly efficient for simple periodic tasks, they do not provide the millisecond-level precision or event persistence offered by SysTimer[1]. By contrast, SysTimer[1] allows thousands of scheduled events with database-backed persistence, making it a better choice for applications requiring reliable, time-based automation. However, for lightweight scheduling tasks on Linux-based

systems, cron jobs remain an efficient alternative due to their minimal overhead.

2. In comparison with RTOS task schedulers, SysTimer[1] offers a more flexible and scalable solution but lacks the hard real-time guarantees necessary for mission-critical applications such as robotic control and real-time data acquisition. RTOS schedulers are optimized for deterministic execution, ensuring that tasks are executed with precise timing. However, they require explicit priority management and manual persistence handling, which can add complexity. SysTimer[1], on the other hand, supports recurring event scheduling (e.g., daily, weekly, monthly) with built-in mechanisms for error handling and recovery, making it a practical alternative for soft real-time applications such as IoT automation and facility management.
3. When compared to hardware timers (MCU-based scheduling mechanisms), SysTimer[1] is significantly more versatile but less precise. Hardware timers operate at microsecond or nanosecond resolution, making them ideal for PWM generation, ADC sampling, motor control, and other high-speed deterministic tasks. However, they lack the ability to handle complex event-based scheduling. Unlike SysTimer[1], hardware timers do not store event information, meaning any scheduled task is lost upon system reset. This makes SysTimer[1] a preferable choice for event-driven automation tasks that require persistence and flexibility, while hardware timers are better suited for applications requiring extreme timing accuracy and deterministic execution.

Overall, SysTimer[1] strikes a balance between precision, scalability, and resource efficiency, making it a versatile alternative to cron jobs and embedded scheduling methods. While RTOS schedulers and hardware timers are better suited for real-time execution, SysTimer[1] provides an optimal solution for periodic event scheduling in domains such as IoT, smart automation, and industrial process management. By incorporating event persistence, scheduling flexibility, and automated error handling, it enhances the reliability of scheduled event execution while maintaining a low computational footprint, making it an excellent choice for applications where timing accuracy, efficiency, and long-term scheduling are key considerations.

7. Conclusion

The SysTimer[1]-based scheduling framework introduced in this paper offers an innovative, efficient and scalable approach to automating tasks based on varying intervals such as Day-to-Day, Week-to-Week, Month-to-Month, and Annual schedules. By leveraging the power of automated triggers and precise timer-based execution, this framework minimizes human intervention and significantly reduces the chances of error. Its flexibility ensures that the system can cater to a broad range of applications, from home automation to industrial control systems, while maintaining reliability and accuracy. Furthermore, the integration of validation mechanisms for handling edge cases, such as leap year considerations and invalid dates, ensures seamless execution of tasks across different scheduling types.

While SysTimer[1] offers millisecond-level precision, it is still a software-based scheduler, which means it can be affected by OS-level latencies. Unlike RTOS task schedulers or hardware timers, it doesn’t provide hard real-time guarantees, making it less suitable for applications that require precise, deterministic execution, such as high- frequency signal processing or ultra-low-latency embedded systems. Another challenge is its reliance on database storage for event persistence, which can introduce performance overhead in memory-constrained environments. Optimizations may be needed to ensure efficient operation on low-power or resource-limited devices. Looking ahead, SysTimer[1] could be enhanced by integrating a hybrid scheduling approach, where hardware timer interrupts work alongside software scheduling to improve precision while maintaining flexibility. Another potential improvement is the use of adaptive scheduling algorithms, which dynamically adjust execution intervals based on system load and real-time

conditions, reducing unnecessary computations and optimizing performance. To further validate its effectiveness, a benchmark comparison with existing scheduling mechanisms—such as cron jobs, RTOS schedulers, and hardware-based timers—would provide deeper insights into its strengths and areas for refinement.

Overall, this framework empowers users to streamline routine processes, increase efficiency, and improve task management, all while ensuring consistency in event execution across varying timeframes. The practical implementation of this framework provides significant benefits in fields like automated system maintenance, scheduling for recurring tasks, and even managing annual events, offering a robust solution for the automation of repetitive processes.

References

[1] Omkar Wagle, “SysTimer:A timer tool for all timer-related applications”, INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH AND CREATIVE TECHNOLOGY, vol. 5, no. 3, pp. 1–4, May 2019, doi: 10.5281/zenodo.14203631.

[2] Kreibich, Jay. Using SQLite. " O'Reilly Media, Inc.", 2010.

[3] Atmoko, Rachmad Andri, Rona Riantini, and Muhammad Khoirul Hasin. "IoT real time data acquisition using MQTT protocol." Journal of Physics: Conference Series. Vol. 853. No. 1. IOP Publishing, 2017.