Systimer: A Timer Tool for All Timer-Related Applications

Omkar Wagle

ov.wagle@gmail.com

Abstract

SysTimer is a memory-efficient tool for embedded systems that optimizes timer array initialization and management. It assigns unique IDs to timers, allowing developers to set expiration times and callback functions. By scanning the timer array and identifying the next timer to expire, SysTimer ensures the timely execution of tasks. The tool's flexible design includes features like time updates and timer cancellations, giving developers greater control over timer behavior. SysTimer's efficient memory usage and versatility make it a valuable asset for embedded systems that require precise timing and resource optimization.

Keywords: Embedded systems, Timer management, callback function, memory optimization, Internet of Things, MQTT

Introduction

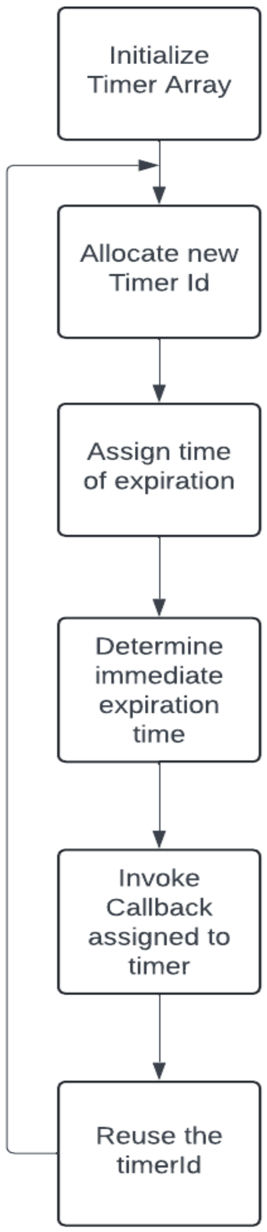
Embedded systems often operate with limited memory resources, necessitating efficient memory management techniques. This paper introduces SysTimer, a tool designed to optimize timer array initialization and management in such environments. By employing a timer ID system and a strategic memory allocation approach, SysTimer ensures that timer-related operations are executed efficiently while minimizing memory consumption.

SysTimer provides a flexible framework for developers to create and manage timers, assigning unique IDs to each timer and allowing for customization of expiration times and callback functions. The tool's ability to efficiently scan the timer array and identify the next expiration time enables timely execution of associated tasks. Additionally, SysTimer offers features like time updates and timer cancellations, providing developers with greater control over timer behavior.

This paper discusses various uses of SysTimer. It discusses how SysTimer helped us to redirect our entire processing into a worker thread and this eventually helped in improving the efficiency of our IoT infrastructure.

Architecture

Given that most embedded systems have very little memory to use, this tool makes sure to initialize the timer array in a multiple of 64. The number 64 was used just for design purposes, however, developers can use any multiples of 8 to initialize the timer array. A unique timer ID is assigned to every timer-related operation.


Initialize Timer Array
Allocate new Timer Id
Assign time of expiration
Determine immediate expiration time
Invoke Callback assigned to timer
Reuse the timerId

During the time of allocation, the developer should mention how many seconds they expect the timer to expire and also the callback function that would be triggered after the timer expires. Once the timer ID is assigned, the timer is added to the timer array and waits for the timer to expire. Multiple timer IDs with different expiration times and callbacks can be assigned in parallel. How does the SysTimer know which timer ID to look for and execute its callback function? SysTimer has a function that lets the application know which timer is near its expiration. It scans through the entire array and finds the smallest expiration time. With this time, the application using this tool can use the time to execute the next timer ID to expire. Once the timer ID is executed and its timer reaches 0, the tool relieves the timer ID for future use. The reason for this is as many embedded systems have memory constraints, reusing as much timer ID as possible before the array expands by 64 is a better way of saving memory. The callback parameter is kept as void * to typecast this parameter into any user-defined type. This design decision was made with the flexibility SysTimer can offer to other developers kept in mind. SysTimer also gives us the provision to update the time or cancel the timer ID. These features help us modify the time or deallocate the timer ID as per the application's wish.

Usage

In today's world, with the modernization of various processors with multiple cores, usage of multithreaded applications has become mandatory mostly for performance enhancements such as performing various operations parallelly with each thread responsible for a set of operations. I have designed an Internet of Things(IoT)[1] application on an ARM9 processor with a single core. Internet of Things is a group of smart devices that collect data and send all that data over the internet to a central hub, it can be a server or a gateway. The IoT application usually receives a command through a remote app, either a mobile app or a website. The command is redirected to the central hub via the internet. The response is received through an HTTP response and is transmitted to a protocol-specific application. This interprocess communication is done through the MQTT[2] protocol.

MQTT stands for Message Queuing Telemetry Transport. MQTT protocol is mostly used for IoT-based interprocess communication. MQTT uses a publish/subscribe model to communicate. The application using MQTT subscribes to a set of topics from where they would want to receive messages. The subscription is registered on the MQTT bus. Once an application publishes a message on any topic, the MQTT bus looks up for which application that message belongs, the MQTT bus then invokes the callback registered to handle all the incoming MQTT messages.

As most of the IoT ecosystems developed today are completely multi-threaded, SysTimer plays an important role in providing callback functionalities with a dedicated timer. When an application receives any command through the MQTT protocol, MQTT invokes the associated callback in its worker thread. The main application generally does not process the incoming command in the same thread from which it received the message. The reason is sometimes, processing a message can take some time and as it is happening sequentially, it blocks the MQTT worker thread. This can cause message losses when the system is put under extreme stress. To avoid this, the application wishes to handle all the processing in its worker thread thereby keeping the MQTT thread as little worked as possible. Keeping the above design in mind, the MQTT thread unloads the complete payload onto the main application worker thread. The main question is how does the main thread start processing this incoming payload? The main worker thread is handled by the SysTimer tool. As per SysTimer design, whenever the timer assigned to a timer ID expires, it invokes the callback assigned to it. When the MQTT thread unloads the payload onto the main worker thread, the main thread assigns a timer ID to it with a zero time to expire. When SysTimer checks for the next time to expire in its array of timer IDs, it notices a timer ID with zero time to expire. This helps to invoke the callback in the context of the main application worker thread. SysTimer helps any application redirect any operation to run in the context of the main application worker thread.

Result

SysTimer led to the development of various tools that became an integral part of our IoT infrastructure. We compared the results of our application with SysTimer and without SysTimer. Without SysTimer, all the MQTT messages were processed in the MQTT thread. We sent around 100 MQTT messages in fast succession. We observed a substantial delay between the sending of commands and its effect on a smart device in the latter commands. The MQTT thread processes the next command only after successfully writing the command to the smart device socket.

This sequential processing in the context of the MQTT thread added up the processing delays. As the MQTT thread is busy, the next incoming command remains in the MQTT queue and hence adds up the processing delays. After performing this experiment for 5 times, we observed that the time the last command was sent and time the smart device responded for that last command was approximately **10 seconds** apart.

SysTimer made sure that all this hassle was avoided. With SysTimer, there were no delays in the MQTT command processing. There was almost no delay in command being sent and the smart device responding to the command. While processing the 100th command, the maximum delay that was observed between the

command being sent to the command reflecting on the smart device was approximately **1 sec**.
The overall improvement with SysTimer was around **90%.** We tried this experiment on a single core processor, hence all the MQTT commands were processed in a sequential manner but it unblocked the MQTT thread and there was no queueing in the MQTT thread.

Future Enhancements

SysTimer, a memory-efficient tool for embedded systems, currently employs an array data structure to store and manage timer IDs. While effective for smaller workloads, the linear search required to find the next timer to expire can introduce processing delays as the number of timers increases. To address this limitation, future iterations of SysTimer could explore the implementation of a priority queue data structure. By organizing timer IDs based on their expiration times, a priority queue would enable constant retrieval of the next timer to expire, significantly improving performance and responsiveness, especially in scenarios with a large number of concurrent timers.

Conclusion

SysTimer has proven to be an invaluable tool in optimizing the performance of IoT applications, particularly those running on single-core processors. By offloading the processing of incoming MQTT messages from the MQTT thread to the main application worker thread, SysTimer effectively prevents the MQTT thread from becoming blocked and ensures timely message processing.
This study demonstrated a significant improvement in application responsiveness and efficiency when SysTimer was used. Compared to the application without SysTimer, which experienced substantial delays in processing MQTT messages, the SysTimer-enabled application exhibited a near-instantaneous response time. This improvement is attributed to SysTimer's ability to efficiently manage task scheduling and execution, ensuring that the MQTT thread remains free to handle incoming messages promptly.
In conclusion, SysTimer is a crucial component for building high-performance IoT applications that rely on MQTT for interprocess communication. Its ability to offload tasks to the main application worker thread and its efficient task scheduling mechanism make it a valuable tool for optimizing performance and ensuring reliable operation in resource-constrained environments.

Reference

1. Rose, Karen, Scott Eldridge, and Lyman Chapin. "The internet of things: An overview." The internet society (ISOC) 80.15 (2015): 1-53.
2. 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.