ANCIT
TECHNICAL NOTE

Using OS Hooks for CPU Load and Memory Usage Monitoring

Technical Note – OS Task and Application Runnable Profiling

Written by
Dr. Kaarthick Balakrishnan
ANCIT | COIMBATORE, INDIA

CPU load and resource consumption measurement is important in AUTOSAR projects to ensure that ECU software operates efficiently and meets real-time performance requirements. It helps engineers identify CPU-intensive tasks and runnables, analyze peak loads, optimize task scheduling, evaluate BSW and cybersecurity overhead, detect performance regressions between software releases, and balance processing across multiple cores.

Typical use cases include ECU and MCU sizing, task and runnable optimization, diagnostic performance measurement, cybersecurity overhead analysis, overload and deadline monitoring, multicore load balancing, and startup/shutdown profiling.

Operating System hooks provide predefined integration points that allow application software to execute custom monitoring or diagnostic code when specific OS events occur. They are useful for profiling task execution without modifying the OS scheduler source code.

1. What Is an OS Hook?

An OS Hook is a function called automatically (Callback function) by the Operating System when a defined event occurs. The OS reserves the integration point, while the application supplies the function body. Some of the most commonly OS Hooks include

OS Event	Typical Hook
Task starts or resumes	PreTaskHook()
Task is stopped, preempted, or terminated	PostTaskHook()
An OS error occurs	ErrorHook()
The OS is shutting down	ShutdownHook()

The key benefit is portability: monitoring and diagnostic functionality can be added through configured hooks rather than by editing the scheduler implementation.

The OS hook options are configured in the **Runtime Systems container**.

2. Measuring CPU Load of an OS Task

CPU load measurement is based on recording how long each task actually occupies the processor.

Without hooks: you'd have to edit the OS scheduler source to insert timing code before/after every task — not allowed, not portable.

With hooks: OS config just enables OsPreTaskHook / OsPostTaskHook. You write the bodies; OS calls them automatically.

Basic sequence: Start timestamp → Task executes → Stop timestamp → Accumulate execution time

2.1 PreTaskHook — Record the Start Time

When a task starts or resumes execution, the OS invokes PreTaskHook(). The application identifies the current task and stores a timestamp.

2.2 PostTaskHook — Calculate the Execution Slice

When the running task is preempted, stopped, or terminated, the OS invokes PostTaskHook(). The elapsed execution time for that slice is calculated and added to the task's accumulated runtime.

Execution Time = End Timestamp − Start Timestamp

2.3 CPU Load Calculation

Over a defined measurement window, the accumulated runtime of each task is divided by the total measurement time.

Task CPU Load (%) = (Total Task Execution Time / Measurement Window) × 100

Example:

Measurement	Value
Measurement Window	100 ms
Task A Runtime	20 ms → 20% CPU load
Task B Runtime	30 ms → 30% CPU load
Idle Time	50 ms → 50% idle
Total CPU Load	50%

> **NOTE**
> *Sum of all tasks' loads + idle-task load + ISR time should equal ~100% of the window — useful as a sanity check that the hook instrumentation isn't losing time (e.g. hook overhead itself, or missed preemption edges).*

3. Handling Task Preemption

Hooks naturally support preemptive scheduling because execution time is accumulated in slices. Consider Task A running for 2 ms before being preempted by Task B. Task B runs for 1 ms and completes, after which Task A resumes and runs for another 3 ms.

Task A runtime: 2 ms + 3 ms = 5 ms

Task B runtime: 1 ms

This gives the actual processor execution time consumed by each task rather than the task's total response time from activation to completion.

4. Measuring CPU Time of an Application Runnable

An OS Task may execute one or more application Runnables. PreTaskHook() and PostTaskHook() measure the execution time of the complete Task; they cannot directly separate the execution time of individual Runnables.

Example task structure:

```
OS Task
|-- Runnable A
|-- Runnable B
`-- Runnable C
```

To profile an individual Runnable, timestamps must be captured immediately before and after its execution. This can be done using manual instrumentation, generated RTE instrumentation, trace mechanisms, or a profiling facility provided by the development environment.

Runnable Execution Time = Runnable End Timestamp − Runnable Start Timestamp

5. Measuring Memory Usage

OS hooks do not directly measure memory consumption. For an OS Task, the most relevant task-specific memory measurement is usually stack usage. A common approach is the stack watermark technique.

5.1 Stack Watermark Method

01	Before execution, fill the allocated task stack with a known pattern such as 0xA5.
02	Allow the task to execute under representative operating conditions.
03	Inspect the stack region and identify how much of the original pattern remains untouched.
04	Calculate the maximum observed stack usage from the overwritten region.

Example: Allocated stack = 2048 bytes; untouched area = 768 bytes; maximum observed stack usage = 1280 bytes.

Hooks can be used as convenient trigger points to record stack-pointer values or initiate periodic checks, but the memory measurement itself comes from stack-pointer monitoring or watermark analysis.

6. Task Profiling vs Runnable Profiling

Objective	Recommended Mechanism	What It Measures
Task CPU load	PreTaskHook() + PostTaskHook()	Complete task execution time
Runnable CPU time	Instrumentation around Runnable execution	Individual Runnable execution time
Task stack usage	Stack watermark / stack pointer monitoring	Maximum observed task stack usage

7. Summary

OS hooks provide a clean and portable mechanism for observing OS events. PreTaskHook() and PostTaskHook() can be used to accumulate execution slices and calculate per-task CPU load without changing the OS scheduler. For application Runnables, separate entry/exit instrumentation is required. Memory usage, especially task stack usage, should be measured using stack watermarking or stack-pointer monitoring; hooks may support the monitoring workflow but are not themselves memory measurement mechanisms.

KEY IDEA

OS Hooks provide the event points; application code provides the measurement logic.

*This technical note is authored by **Dr. Kaarthick Balakrishnan** and published by **ANCIT** — a leading automotive engineering training company specialising in AUTOSAR, Embedded Systems, Functional Safety, and Software-Defined Vehicles. For training enquiries, contact us at **beeshma@ancitconsulting.com** or call **+91 9840378602**.*