STAY AHEAD OF GAME USING RTOS

Aman Kanwar

AGENDA

- FreeRTOS Tracing Tool integration
- FreeRTOS task state
- FreeRTOS task state transition
- Task Scheduling
- Task Parameters
- Task Handle
- Basic Task Management
- Quick Test

WHAT IS THE RUNTIME?

- For all the problem statements, how will you conclude on the execution time?
- What is the execution time for a task, which task is running first?

SYSTEM TRACE VIEWER

- System trace viewers are tools for analyzing and debugging real-time embedded systems.
- They provide insights into task scheduling, interrupt handling, and system behavior.
- Trace viewers help identify performance bottlenecks, optimize code, and troubleshoot issues.
- Optimize code how? Ideas?
- They are essential for understanding and improving the behavior of complex embedded applications.

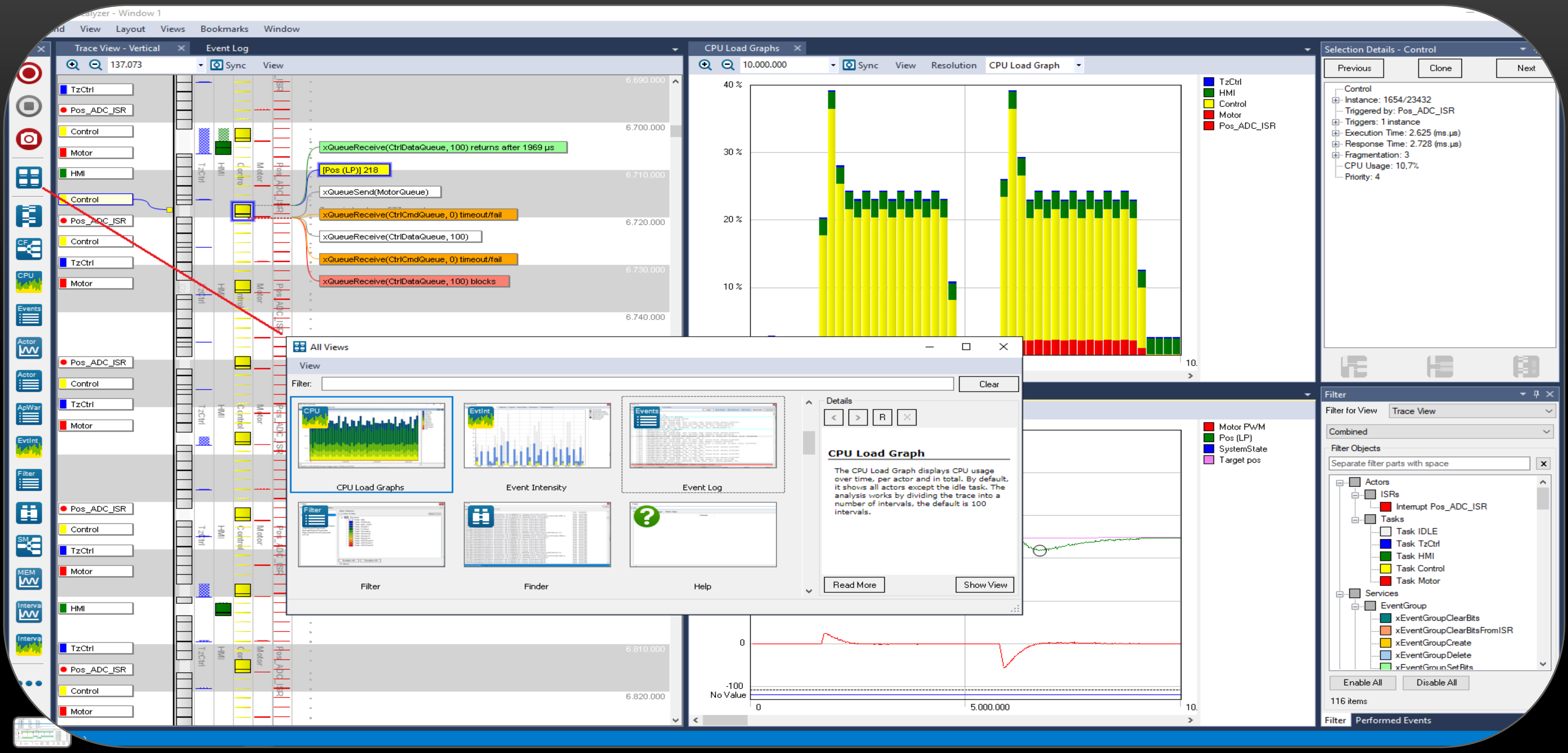
Available options

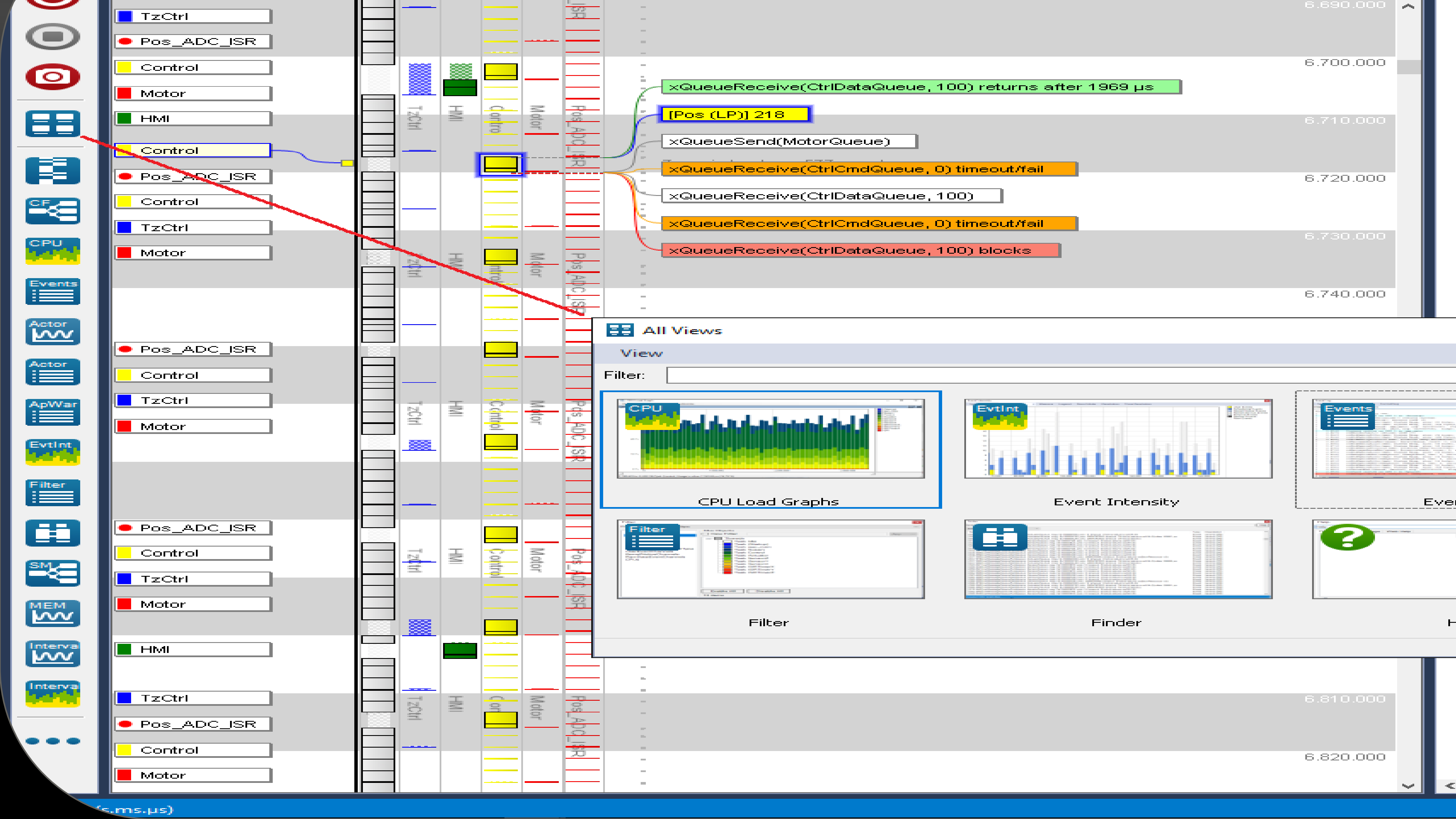
- Segger SystemView
- Percepio Tracealyzer
- Lauterbach TRACE32
- Renesas e2 studio with RTT
- TraceLink by AdaCore
- ULINK and ULINKplus by Keil (Arm)

Tracing Tools

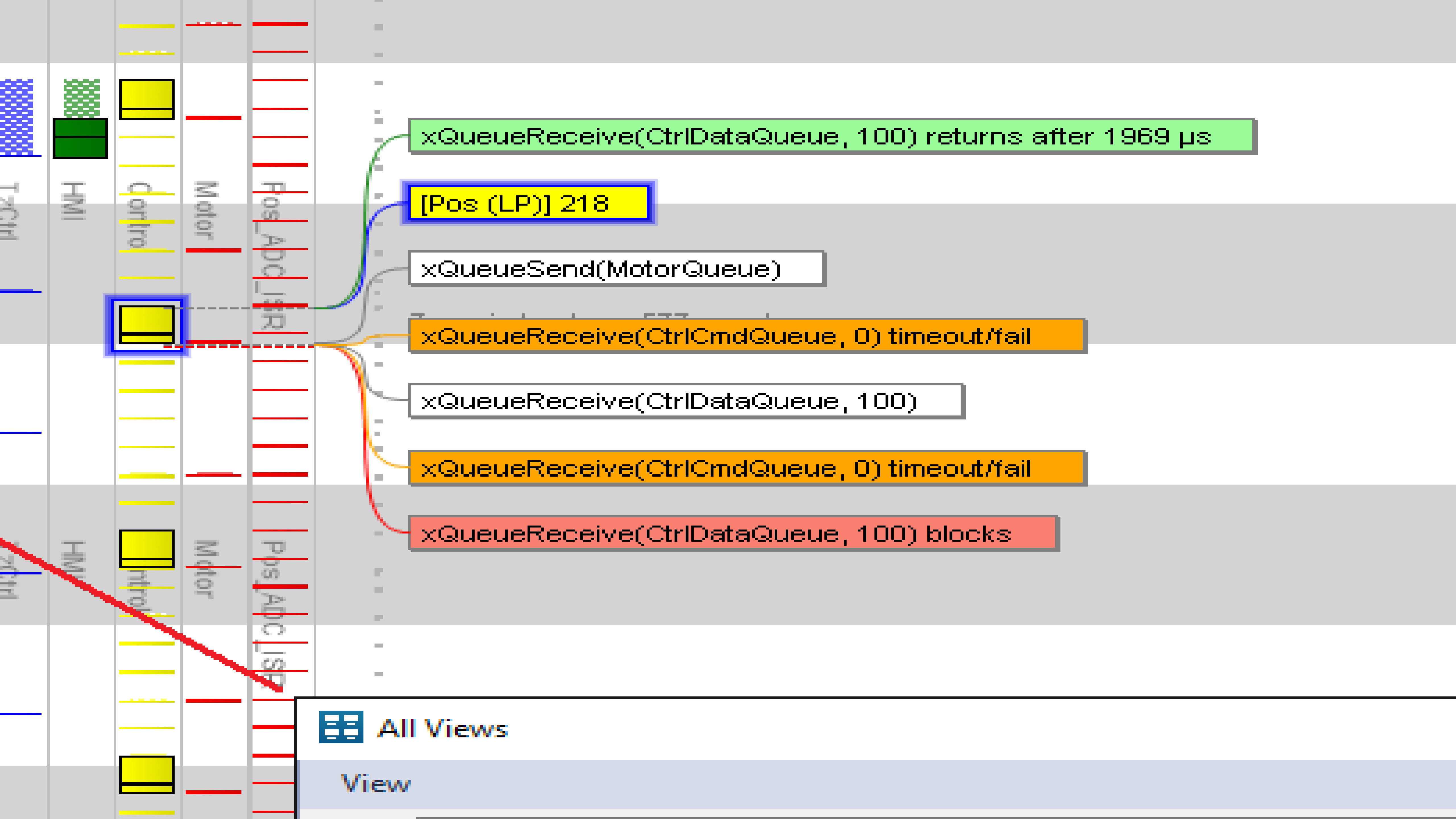
percipio®

Powered by Skillsoft®

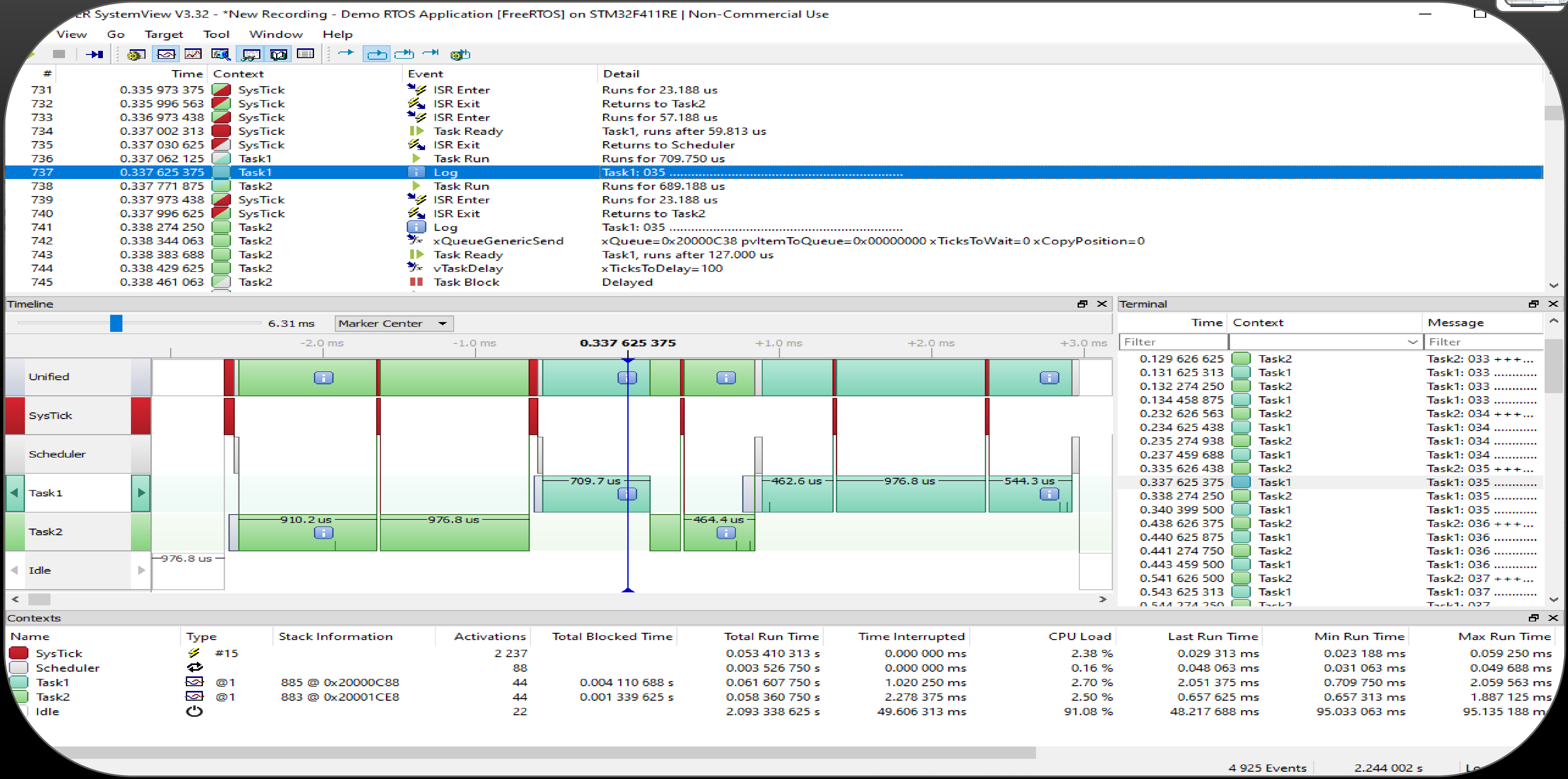

Trace View - Vertical
Event Log
CPU Load Graphs
Selection Details - Control
Previous
Clone
Next
Control
Instance: 1654/23432
Triggered by: Pos_ADC_ISR
Triggers: 1 instance
Execution Time: 2.625 (ms.µs)
Response Time: 2.728 (ms.µs)
Fragmentation: 3
CPU Usage: 10,7%
Priority: 4
xQueueReceive(CtrlDataQueue, 100) returns after 1969 µs
[Pos (LP)] 218
xQueueSend(MotorQueue)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100) blocks
All Views
Filter:
Clear
Details
CPU Load Graph
The CPU Load Graph displays CPU usage over time, per actor and in total. By default, it shows all actors except the idle task. The analysis works by dividing the trace into a number of intervals, the default is 100 intervals.
Read More
Show View
CPU Load Graphs
Event Intensity
Event Log
Filter
Finder
Help
TzCtrl
HMI
Control
Motor
Pos_ADC_ISR
Motor PWM
Pos (LP)
SystemState
Target pos
Filter
Filter for View
Trace View
Combined
Filter Objects
Separate filter parts with space
Actors
ISRs
Interrupt Pos_ADC_ISR
Tasks
Task IDLE
Task TzCtrl
Task HMI
Task Control
Task Motor
Services
EventGroup
xEventGroupClearBits
xEventGroupClearBitsFromISR
xEventGroupCreate
xEventGroupDelete
Enable All
Disable All
116 items
Performed Events

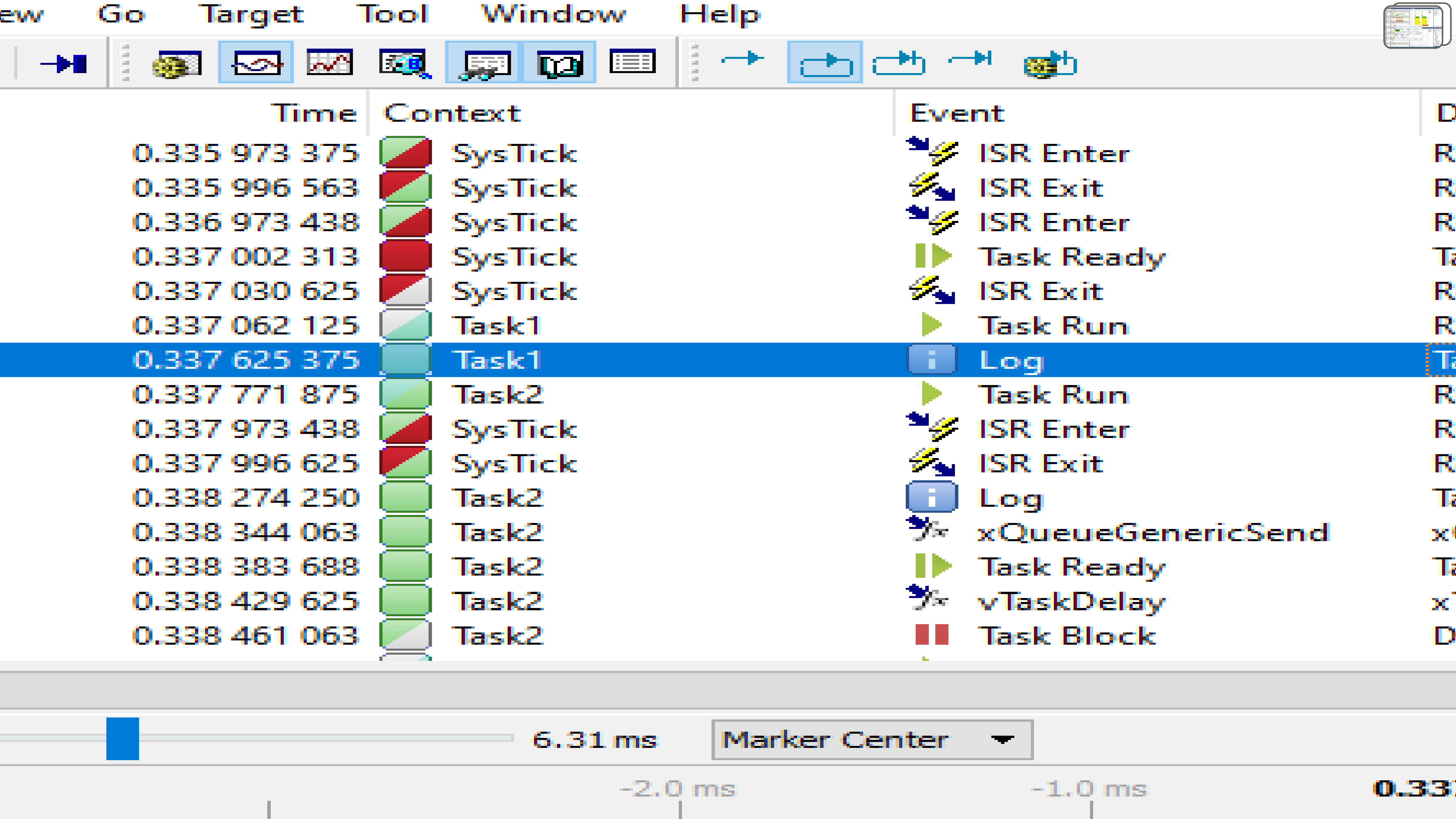


TzCtrl
Pos_ADC_ISR
Control
Motor
HMI
CF
CPU
Events
Actor
ApWar
EvtInt
Filter
SM
MEM
Interva
xQueueReceive(CtrlDataQueue, 100) returns after 1969 µs
[Pos (LP)] 218
xQueueSend(MotorQueue)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100) blocks
6.690.000
6.700.000
6.710.000
6.720.000
6.730.000
6.740.000
6.810.000
6.820.000
All Views
View
Filter:
CPU Load Graphs
Event Intensity
Filter
Finder
(s.ms.µs)

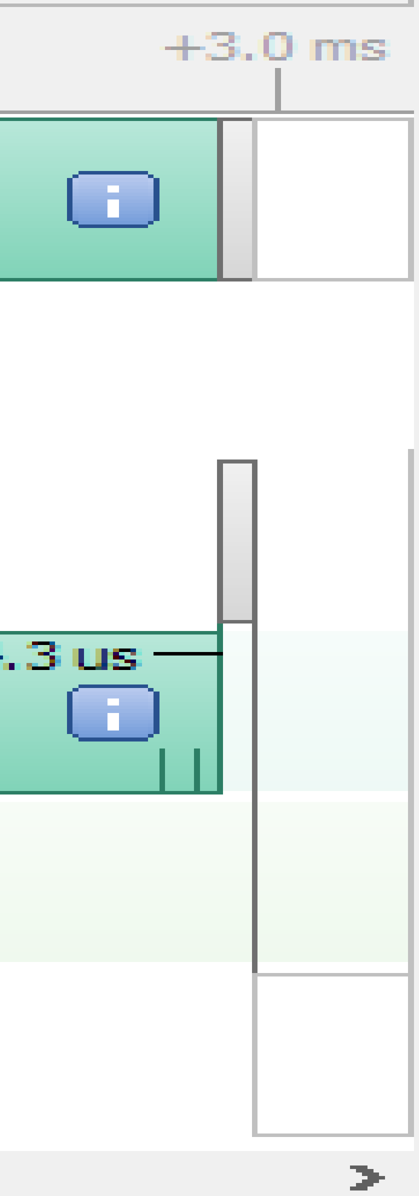

xQueueReceive(CtrlDataQueue, 100) returns after 1969 µs
[Pos (LP)] 218
xQueueSend(MotorQueue)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100)
xQueueReceive(CtrlCmdQueue, 0) timeout/fail
xQueueReceive(CtrlDataQueue, 100) blocks
HMI
Motor
All Views
View

SystemView

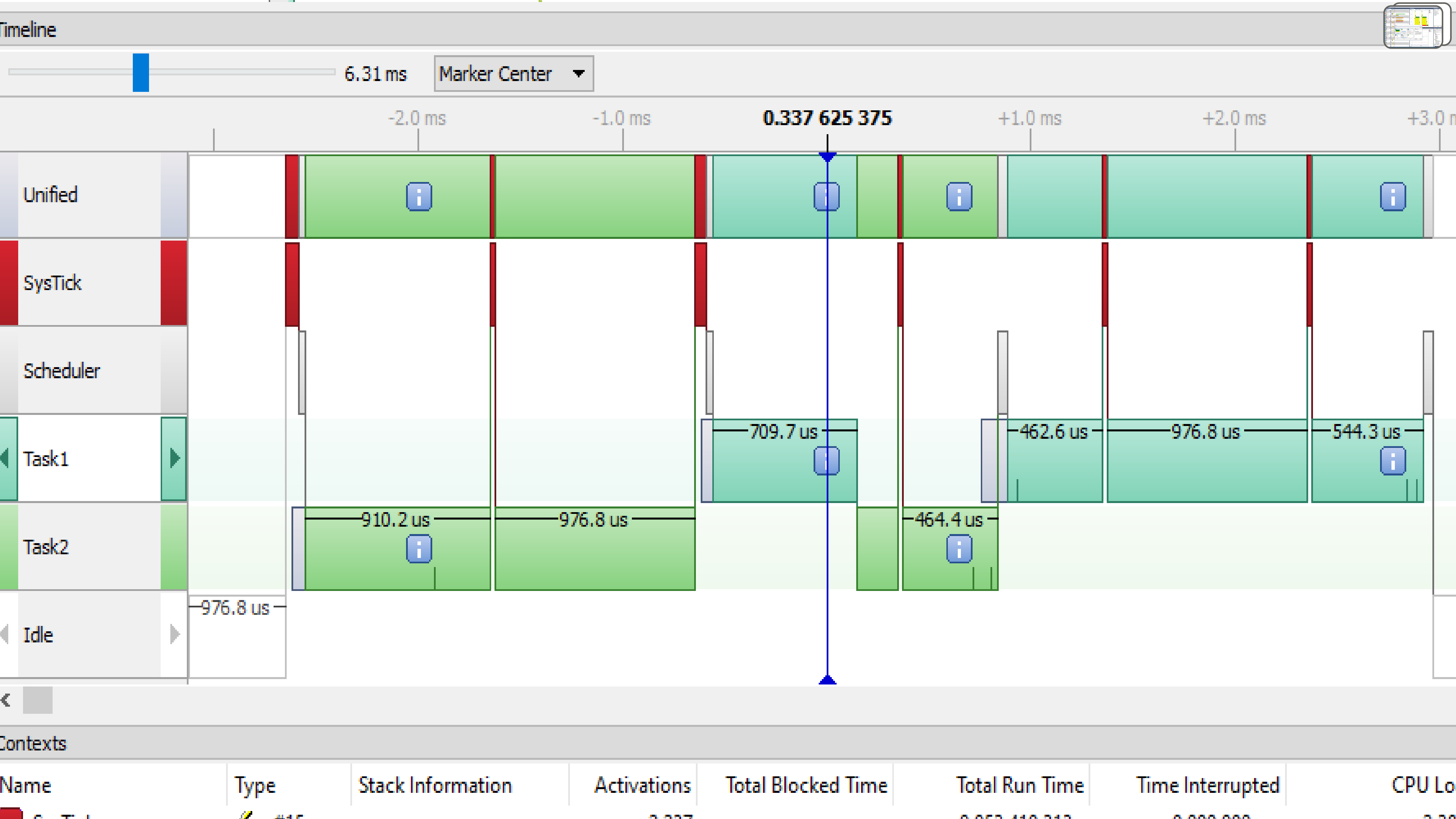

ER SystemView V3.32 - *New Recording - Demo RTOS Application [FreeRTOS] on STM32F411RE | Non-Commercial Use
View Go Target Tool Window Help
Time Context Event Detail
731 0.335 973 375 SysTick ISR Enter Runs for 23.188 us
732 0.335 996 563 SysTick ISR Exit Returns to Task2
733 0.336 973 438 SysTick ISR Enter Runs for 57.188 us
734 0.337 002 313 SysTick Task Ready Task1, runs after 59.813 us
735 0.337 030 625 SysTick ISR Exit Returns to Scheduler
736 0.337 062 125 Task1 Task Run Runs for 709.750 us
737 0.337 625 375 Task1 Log Task1: 035 ....
738 0.337 771 875 Task2 Task Run Runs for 689.188 us
739 0.337 973 438 SysTick ISR Enter Runs for 23.188 us
740 0.337 996 625 SysTick ISR Exit Returns to Task2
741 0.338 274 250 Task2 Log Task1: 035 ....
742 0.338 344 063 Task2 xQueueGenericSend xQueue=0x20000C38 pvItemToQueue=0x00000000 xTicksToWait=0 xCopyPosition=0
743 0.338 383 688 Task2 Task Ready Task1, runs after 127.000 us
744 0.338 429 625 Task2 vTaskDelay xTicksToDelay=100
745 0.338 461 063 Task2 Task Block Delayed
Timeline
6.31 ms Marker Center
-2.0 ms -1.0 ms 0.337 625 375 +1.0 ms +2.0 ms +3.0 ms
Unified
SysTick
Scheduler
Task1
Task2
Idle
709.7 us 462.6 us 976.8 us 544.3 us
910.2 us 976.8 us 464.4 us
976.8 us
Terminal
Time Context Message
Filter Filter
0.129 626 625 Task2 Task2: 033 ++++...
0.131 625 313 Task1 Task1: 033 ....
0.132 274 250 Task2 Task1: 033 ....
0.134 458 875 Task1 Task1: 033 ....
0.232 626 563 Task2 Task2: 034 ++++...
0.234 625 438 Task1 Task1: 034 ....
0.235 274 938 Task2 Task1: 034 ....
0.237 459 688 Task1 Task1: 034 ....
0.335 626 438 Task2 Task2: 035 ++++...
0.337 625 375 Task1 Task1: 035 ....
0.338 274 250 Task2 Task1: 035 ....
0.340 399 500 Task1 Task1: 035 ....
0.438 626 375 Task2 Task2: 036 ++++...
0.440 625 875 Task1 Task1: 036 ....
0.441 274 750 Task2 Task1: 036 ....
0.443 459 500 Task1 Task1: 036 ....
0.541 626 500 Task2 Task2: 037 ++++...
0.543 625 313 Task1 Task1: 037 ....
Contexts
Name Type Stack Information Activations Total Blocked Time Total Run Time Time Interrupted CPU Load Last Run Time Min Run Time Max Run Time
SysTick #15 2 237 0.053 410 313 s 0.000 000 ms 2.38 % 0.029 313 ms 0.023 188 ms 0.059 250 ms
Scheduler 88 0.003 526 750 s 0.000 000 ms 0.16 % 0.048 063 ms 0.031 063 ms 0.049 688 ms
Task1 @1 885 @ 0x20000C88 44 0.004 110 688 s 0.061 607 750 s 1.020 250 ms 2.70 % 2.051 375 ms 0.709 750 ms 2.059 563 ms
Task2 @1 883 @ 0x20001CE8 44 0.001 339 625 s 0.058 360 750 s 2.278 375 ms 2.50 % 0.657 625 ms 0.657 313 ms 1.887 125 ms
Idle 22 2.093 338 625 s 49.606 313 ms 91.08 % 48.217 688 ms 95.033 063 ms 95.135 188 m
4 925 Events 2.244 002 s

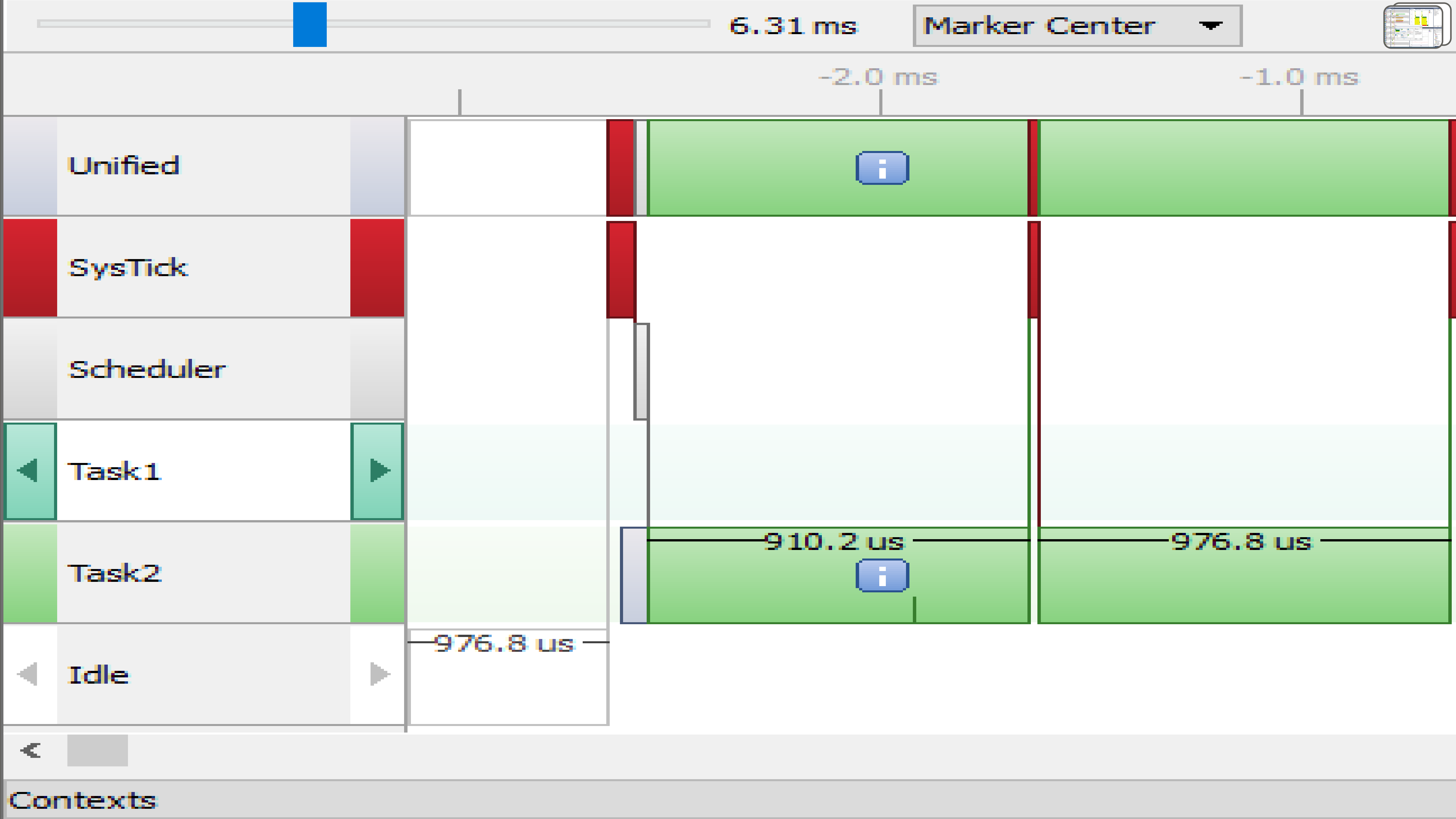


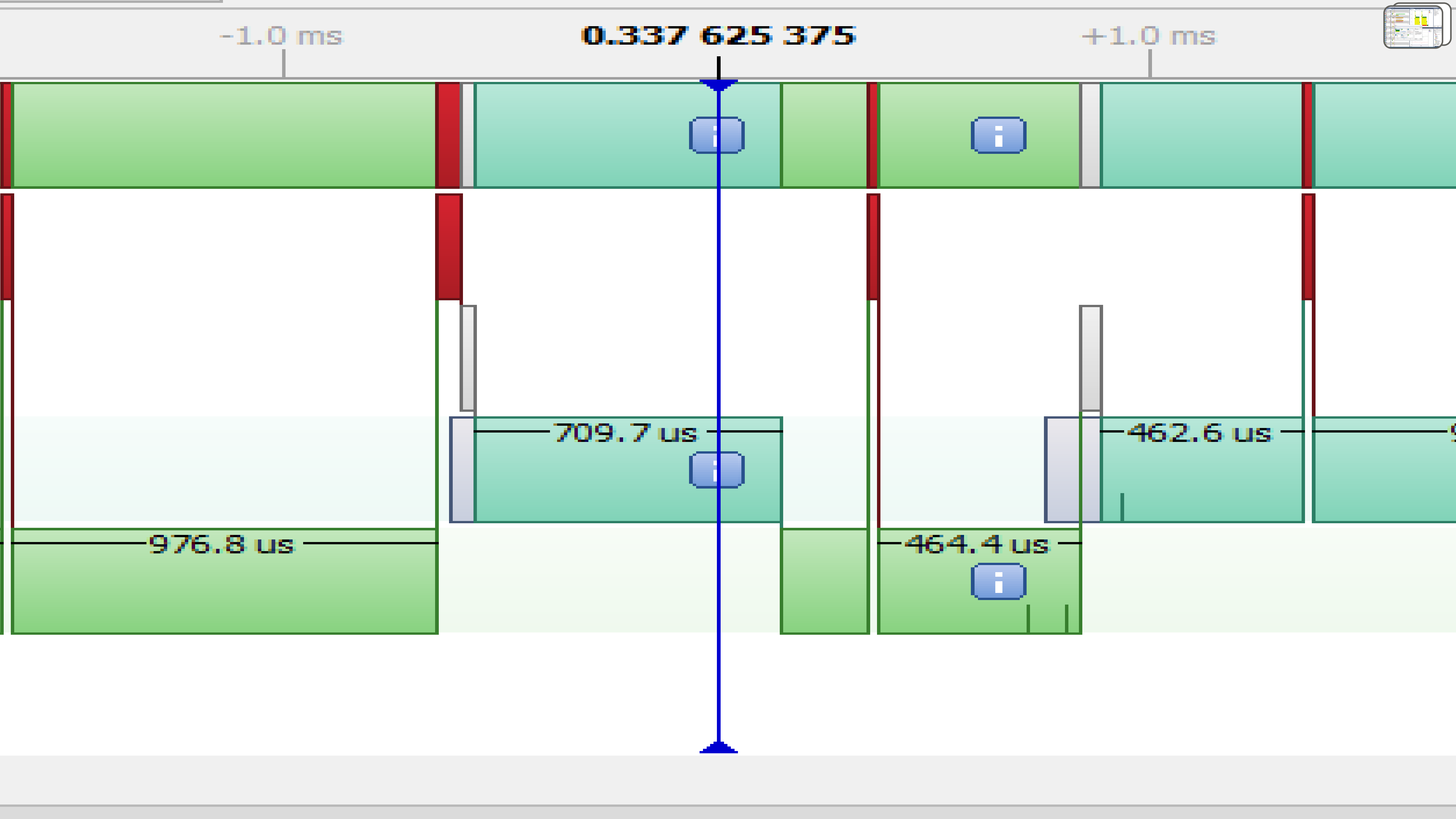
ew Go Target Tool Window Help
Time Context Event
0.335 973 375 SysTick ISR Enter
0.335 996 563 SysTick ISR Exit
0.336 973 438 SysTick ISR Enter
0.337 002 313 SysTick Task Ready
0.337 030 625 SysTick ISR Exit
0.337 062 125 Task1 Task Run
0.337 625 375 Task1 Log
0.337 771 875 Task2 Task Run
0.337 973 438 SysTick ISR Enter
0.337 996 625 SysTick ISR Exit
0.338 274 250 Task2 Log
0.338 344 063 Task2 xQueueGenericSend
0.338 383 688 Task2 Task Ready
0.338 429 625 Task2 vTaskDelay
0.338 461 063 Task2 Task Block
6.31 ms
Marker Center
-2.0 ms
-1.0 ms



Time	Context	Message
Filter		Filter
0.129 626 625	Task2	Task2: 033 +++...
0.131 625 313	Task1	Task1: 033 ....
0.132 274 250	Task2	Task1: 033 ....
0.134 458 875	Task1	Task1: 033 ....
0.232 626 563	Task2	Task2: 034 +++...
0.234 625 438	Task1	Task1: 034 ....
0.235 274 938	Task2	Task1: 034 ....
0.237 459 688	Task1	Task1: 034 ....
0.335 626 438	Task2	Task2: 035 +++...
0.337 625 375	Task1	Task1: 035 ....
0.338 274 250	Task2	Task1: 035 ....
0.340 399 500	Task1	Task1: 035 ....
0.438 626 375	Task2	Task2: 036 +++...
0.440 625 875	Task1	Task1: 036 ....
0.441 274 750	Task2	Task1: 036 ....
0.443 459 500	Task1	Task1: 036 ....
0.541 626 500	Task2	Task2: 037 +++...
0.543 625 313	Task1	Task1: 037 ....
0.544 274 250	Task2	Task1: 037


Timeline
6.31 ms
Marker Center
-2.0 ms
-1.0 ms
0.337 625 375
+1.0 ms
+2.0 ms
Unified
SysTick
Scheduler
Task1
Task2
Idle
709.7 us
462.6 us
976.8 us
544.3 us
910.2 us
976.8 us
464.4 us
976.8 us
Contexts
Name
Type
Stack Information
Activations
Total Blocked Time
Total Run Time
Time Interrupted


6.31 ms
Marker Center
-2.0 ms
-1.0 ms
Unified
SysTick
Scheduler
Task1
Task2
Idle
910.2 us
976.8 us
976.8 us
Contexts



-1.0 ms
0.337 625 375
+1.0 ms
709.7 us
462.6 us
976.8 us
464.4 us

Tracing Tools

Which one Do you like?

percipio
Powered by Skillsoft.

SystemView

SYSTEM VIEWER INTEGRATION

- A real-time system analysis tool by Segger Microcontroller.
- Offers visualization of task scheduling, interrupt activity, and custom events.
- Supports various microcontrollers and RTOSes.

ASSIGNMENT 1

RECAP ON PREVIOUS PROBLEM STATEMENT
USE TRACING TOOL TO OBSERVE THE BEHAVIOR

PROBLEM 1

Blink LED1 at every 2 mS.

PROBLEM 2

Blink LED1 at every 2 mS

Blink LED2 at every 3 mS

PROBLEM 3

Blink LED1 at every 2 mS

Blink LED2 at every 3 mS

Blink LED3 at every 4 mS

Blink LED3 at every 5 mS

RTOS TASK STATE

Description

FreeRTOS tasks can exist in various states during their execution. Understanding these task states is crucial for managing concurrent tasks in an embedded system.

Ready State (R)

- In this state, a task is ready to run but hasn't been scheduled to execute by the FreeRTOS scheduler.
- Tasks in the ready state are eligible to run, but they are waiting for their turn to execute based on their priority and scheduling algorithm.

Running State (X)

- The running state represents the task that is currently executing on the CPU core.
- At any given time, only one task can be in the running state on a specific CPU core.
- The running task's code is actively executing, and it will continue until it either voluntarily yields the CPU or is preempted by a higher-priority task.

Blocked State (B)

- Tasks in the blocked state are not ready to run, and they are temporarily unable to execute.
- Tasks can enter the blocked state for various reasons, such as waiting for a resource or synchronization event to occur.
- When a task is blocked, it does not consume CPU time and remains inactive until the blocking condition is resolved.

Suspended State (S)

- A task in the suspended state is explicitly halted by the application or another task.
- Suspended tasks do not participate in task scheduling until they are resumed.
- Suspending a task can be useful for temporarily deactivating it without deleting it.

RTOS TASK STATE

Deleted State (D)

- This is not a standard task state.
- When a task is deleted (using vTaskDelete()), it enters the deleted state.
- In this state, the task's resources are released, and its memory is deallocated.
- Deleted tasks can no longer be scheduled or interacted with.

Invalid State (I)

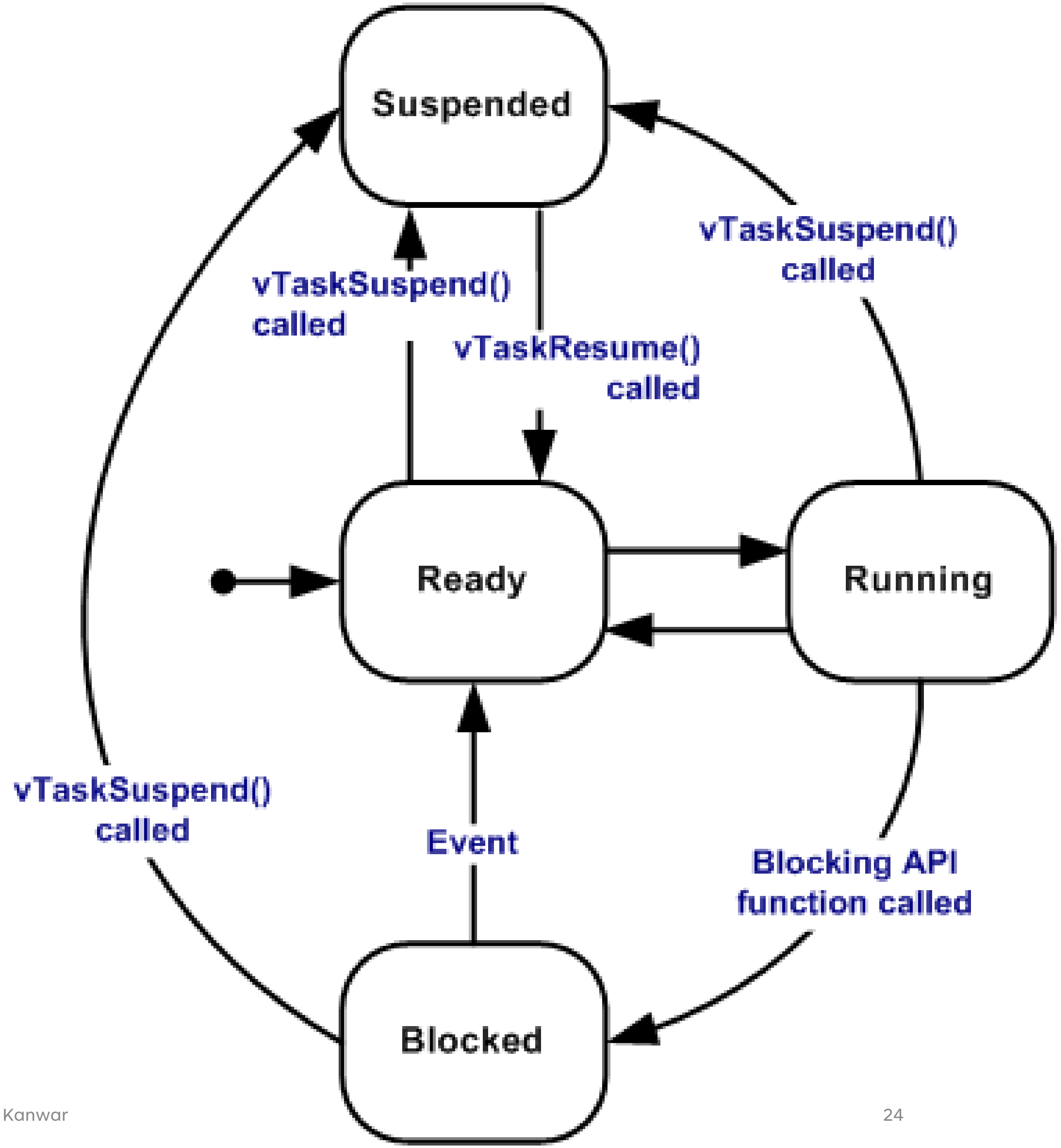
- The invalid state is not a standard task state but may be used in some special scenarios.
- Tasks may enter an invalid state when they have not been properly initialized or when their control structures are corrupted.

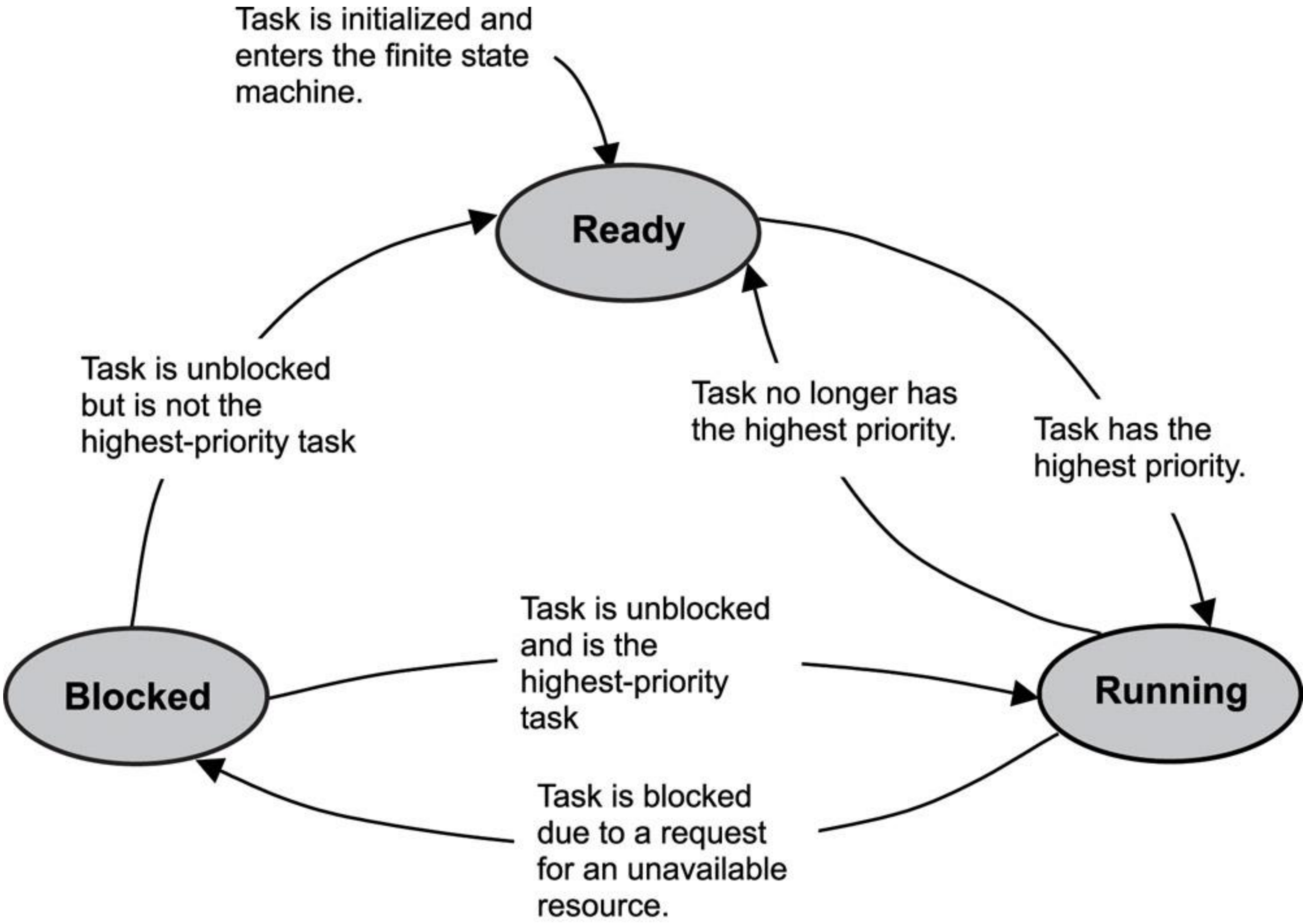
RTOS TASK STATE

- Tasks are created using **xTaskCreate()**.
- A task starts in the ready state after being created.
- When the scheduler starts using **vTaskStartScheduler().**
- Tasks transition from ready to the running state based on priority and scheduling decisions.
- Tasks can voluntarily transition from the running to the ready state via functions like **taskYIELD().**
- Tasks can transition from the running to the blocked state via functions like **vTaskDelay()**.
- Tasks can move from the ready state to the blocked state when they are waiting for events or resources using synchronization mechanisms like semaphores, mutexes, or queues.
- Tasks can be suspended by **vTaskSuspend()**, transitioning from any state to the suspended state.
- Deleted tasks transition to the deleted state when **vTaskDelete()** is called.

TASK STATE TRANSITIONS

TASK STATE TRANSITIONS




Task is initialized and enters the finite state machine.
Ready
Task is unblocked but is not the highest-priority task
Task no longer has the highest priority.
Task has the highest priority.
Blocked
Task is unblocked and is the highest-priority task
Running
Task is blocked due to a request for an unavailable resource.

Assignment Problem 2

Create and RTOS application with following behaviour.

- Create Task1 with priority 1 for toggling LED1, LED2 at every 4 mS
- Create Task2 with priority 2 for toggling LED3, LED4 at every 5 mS
- Use SystemViewer to observe the behaviour
- Start scheduler, observe the behaviour and document.

Scheduling in FreeRTOS

- The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.
- FreeRTOS primarily uses a priority-based preemptive scheduling algorithm.
- Tasks with higher priorities will preempt tasks with lower priorities.

Refer to 3.12 section "Scheduling Algorithms" from RTOS document from here.

TASK SCHEDULING IN FREERTOS

Task Scheduling in FreeRTOS

Task scheduling is a fundamental aspect of FreeRTOS, allowing it to manage the execution of multiple tasks concurrently. FreeRTOS provides a real-time operating system that ensures deterministic and predictable task scheduling.

Scheduling Algorithm

- FreeRTOS uses a priority-based preemptive scheduling algorithm.
- The task with the highest priority that is ready to run gets CPU time.
- Preemption ensures that tasks with higher priority always execute when they are ready.
- Lower-priority tasks run only when no higher-priority tasks are ready.

Scheduler

The scheduler is responsible for deciding which task runs next. FreeRTOS uses a priority-based preemptive scheduling algorithm. Higher-priority tasks preempt lower-priority tasks.

Context Switching

- Context switching is the process of saving the current task's context and loading the context of the next task.
- Occurs when a higher-priority task becomes ready to run or when the currently running task voluntarily yields control.

TASK Scheduling

Description

The preemptive nature ensures that the task with the highest priority that is ready to run will always be the one executing.

Priority-based Scheduling

FreeRTOS uses priority levels to determine the order in which tasks are executed. Lower numerical values represent higher priorities.

The scheduler always selects the highest-priority task that is ready to run.

Preemption

Higher-priority tasks can preempt lower-priority tasks. When a higher-priority task becomes ready to run, it will immediately preempt the currently running task.

Round Robin Scheduling

While the primary scheduling mechanism is priority-based, FreeRTOS also provides a time-slicing feature for tasks with equal priorities. This allows tasks of the same priority to take turns executing in a round-robin fashion.

TASK Scheduling

Configurable Scheduler

FreeRTOS allows for some configuration of the scheduler behavior, providing options to adapt to different application requirements.

Task Yielding

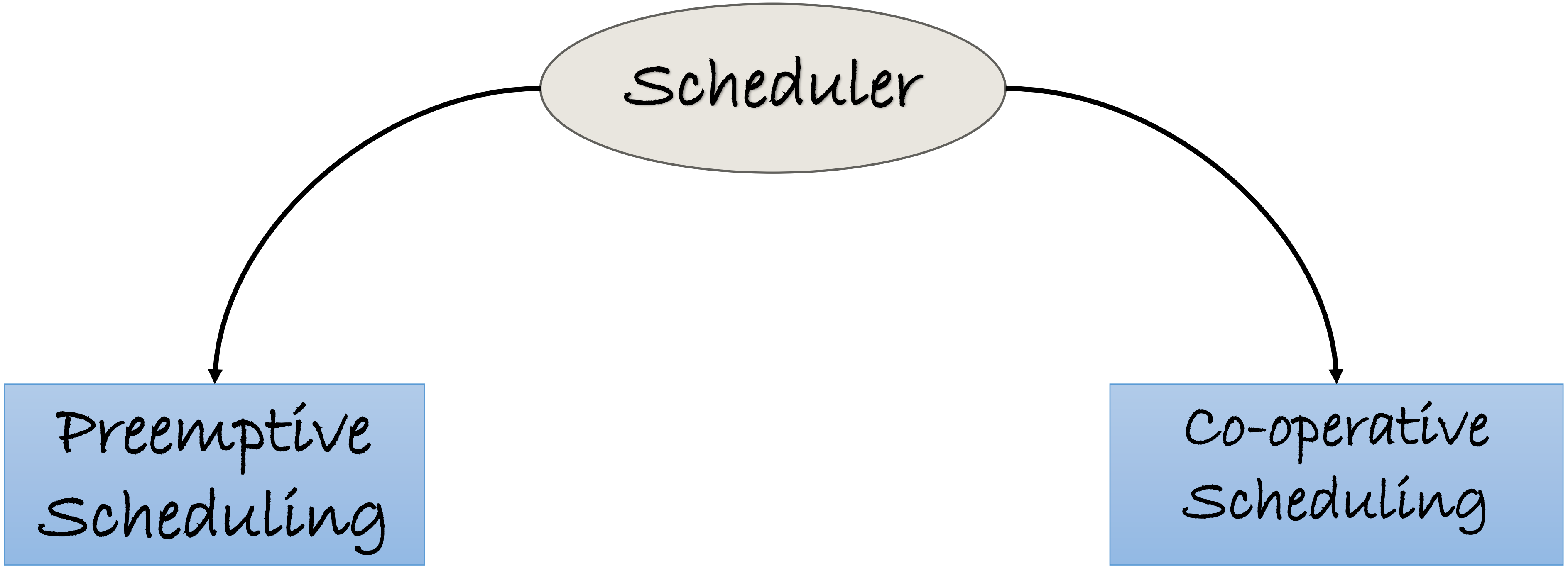
Tasks can explicitly yield the processor using the taskYIELD function, allowing lower-priority tasks to run. This is useful in situations where cooperative multitasking is needed.

Task Suspend/Resume

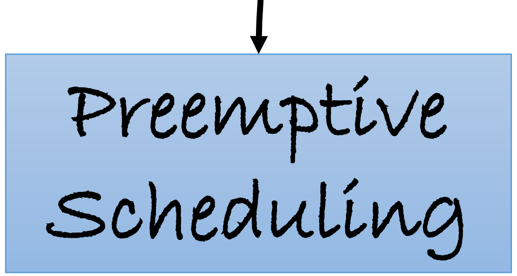
Tasks can be suspended and later resumed, allowing for more fine-grained control over which tasks are allowed to execute.

Idle Task

FreeRTOS includes an idle task, which runs when no other tasks are ready to execute. The idle task typically performs low-priority background activities and helps minimize power consumption.

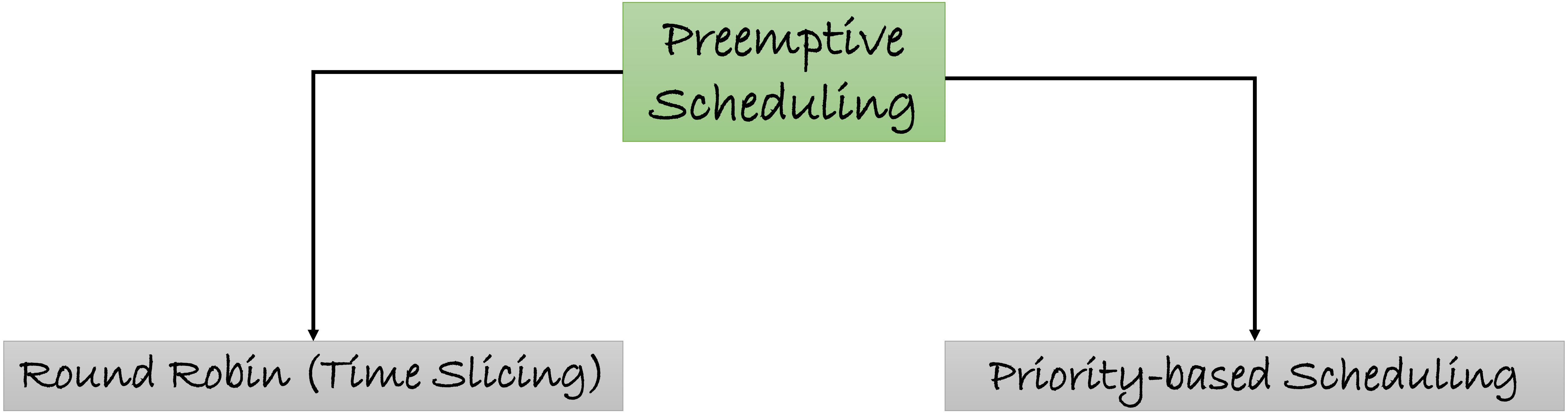


config USE_PREEMPTION is defined in FreeRTOSConfig.h, this macro controls the scheduling policy that RTOS port is going to use. Default is set to 1.

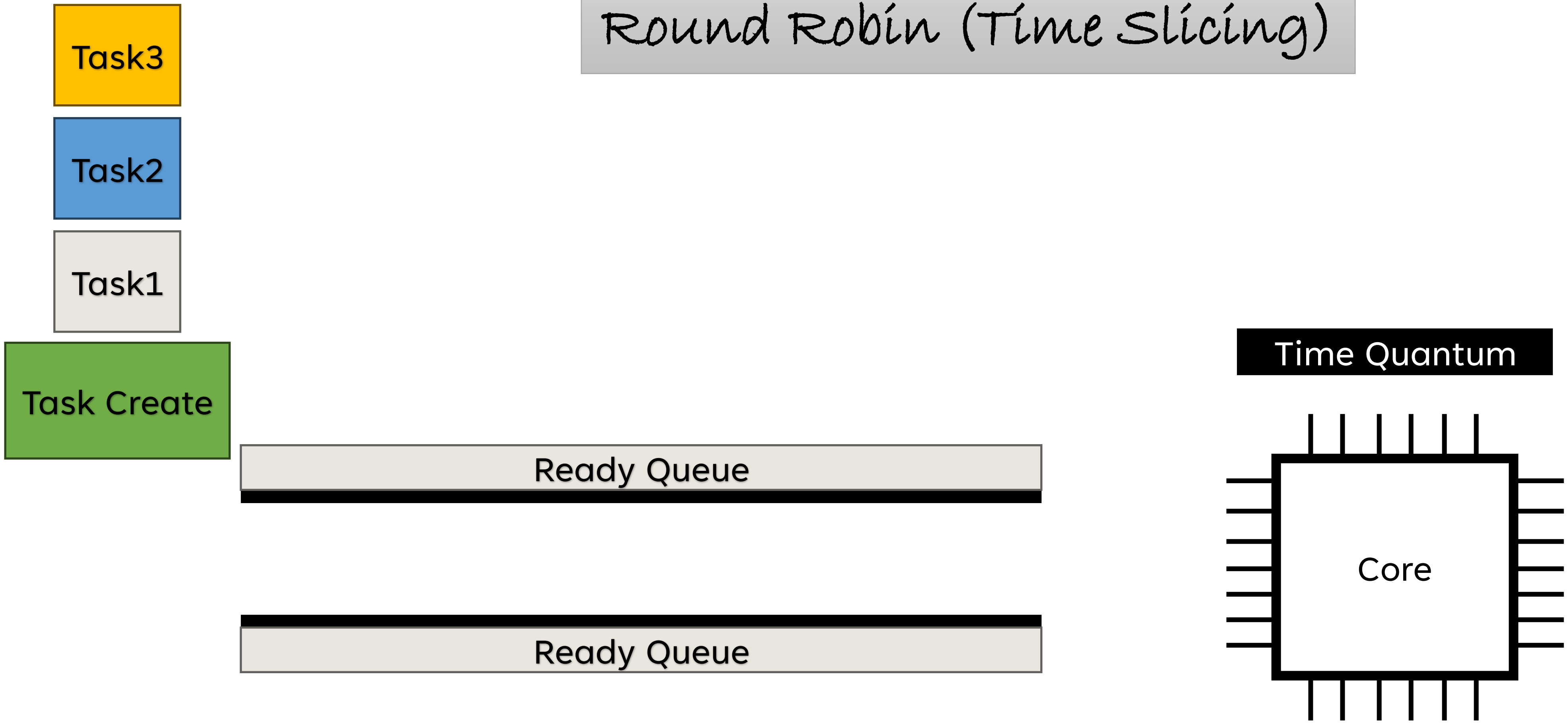


Preemption

Replacing a running task with another task ready to be executed. During preemption the running task is made to give up the processor even if it hasn't finished its work.



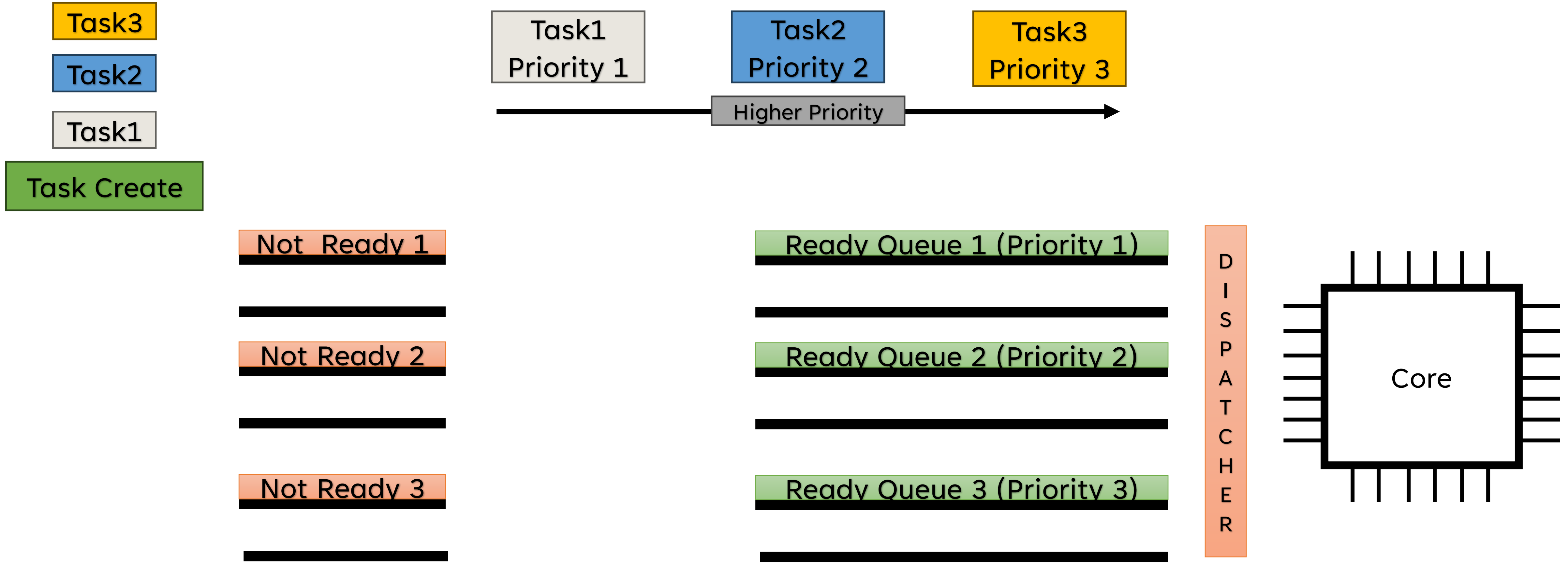
Round Robin (Time Slicing)



Here "Time Quantum" is configurable value in FreeRTOSConfig.h

Aman Kanwar

Priority-based Scheduling



Higher Priority task has more importance!

Whenever any higher priority task is in “Ready” state, it should Run.

Aman Kanwar

PRIORITIZED PRE-EMPTIVE PREEMPTIVE MULTITASKING

- Preemptive multitasking is a form of multitasking where the operating system interrupts and switches between tasks, or threads, without the tasks' cooperation.
- Along with Prioritized preemption, time slicing is also implemented.
- In preemptive multitasking systems, the operating system has the ability to forcefully interrupt a currently running task and switch to another task, typically based on priority and time-sharing considerations.
- Refer to Preemptive Scheduling in Mastering the FreeRTOS™ Real Time Kernel page 92.

COOPERATIVE MULTITASKING

- FreeRTOS primarily employs preemptive scheduling, it also supports cooperative multitasking to some extent.
- Cooperative multitasking means that tasks voluntarily yield control to allow other tasks to run, rather than being preempted by the scheduler.
- Refer to Co-operative Scheduling in Mastering the FreeRTOS™ Real Time Kernel page 98.

Prioritized Pre-emptive Scheduling with Time Slicing and Co-operative Scheduling

- Refer to above topics in Mastering the FreeRTOS™ Real Time Kernel page 92 onwards.
- 10 Mins of Reading ☺
- Can ask random students to come and explain the same ☺

Assignment Problem 3

Scenario 1 (Pre-emptive Scheduling)

Create and RTOS application with following behaviour.

- Create Task1 with priority 2 for toggling LED1.
- Create Task2 with priority 2 for toggling LED2.
- Start scheduler, observe the behaviour and document.

Scenario 2 (Co-operative Scheduling)

Create and RTOS application with following behaviour.

- Set **#define configUSE_PREEMPTION 0 in FreeRTOSConfig.h**
- Create Task1 with priority 2 for toggling LED1.
- Create Task2 with priority 2 for toggling LED2.
- Start scheduler, observe the behaviour and document.

COOPERATIVE MULTITASKING

- Task Yielding()

  Tasks can explicitly yield the processor using the taskYIELD function. When a task calls taskYIELD, it voluntarily gives up its CPU time, allowing other tasks of equal or higher priority to run.

- Cooperative Blocking

  Tasks can be designed to cooperatively block themselves using functions like vTaskSuspend or vTaskDelay. When a task is blocked, it allows other tasks to run until it becomes unblocked.

- Task Notification

  Task notifications can also be used for cooperative scheduling. A task can wait for a notification and voluntarily unblock when the notification is received.

Assignment Problem 4

Scenario 1

Create and RTOS application with following behaviour.

- Create Task1 with priority 2 for toggling LED1 at every 100 ms.
- Create Task2 with priority 2 for toggling LED2 at every 500 ms.
- Start scheduler, observe the behaviour and document.

Scenario 2

Create and RTOS application with following behaviour.

- Set **#define configUSE_PREEMPTION 0 in FreeRTOSConfig.h**
- Create Task1 with priority 2 for toggling LED1 at every 100 ms.
- Create Task2 with priority 2 for toggling LED2 at every 500 ms.
- Start scheduler, observe the behaviour and document.

Assignment Problem 5

Scenario 1

Create and RTOS application with following behaviour.

- Set **#define configUSE_PREEMPTION 0 in FreeRTOSConfig.h**
- Create Task1 with priority 2 for toggling LED1 at every 100 ms and call **taskYIELD();**
- Create Task2 with priority 2 for toggling LED2 at every 500 ms and call **taskYIELD();**
- Start scheduler, observe the behaviour and document.

Scenario 2

Create and RTOS application with following behaviour.

- Set **#define configUSE_PREEMPTION 0 in FreeRTOSConfig.h**
- Create Task1 with priority 1 for toggling LED1 at every 100 ms and call **taskYIELD();**
- Create Task2 with priority 2 for toggling LED2 at every 500 ms and call **taskYIELD();**
- Start scheduler, observe the behaviour and document.

BASIC TASK MANAGEMENT

- Using Task Parameters
- Using Taks Handles

TASK PARAMETERS **IMP**

Description

- Task parameters are used to define certain characteristics and behaviors.

Task Initialization Data

- It is used to pass data or configuration parameters to the task when it starts executing.
- **The task function can access and interpret this data as needed. (Imp)**

Flexibility

- void pointer, pvParameters provides flexibility in the type of data that can be passed.
- Developers can pass a single value, a structure, or dynamically allocated memory, depending on the task's requirements

Task-Specific Configuration

- Allows each instance of a task to have specific configuration or data, making tasks more versatile and reusable.
- The task function must typecast pvParameters to the appropriate data type before using it.
- For example: int myParam = *((int*)pvParameters);
- Instead of relying on global variables, pvParameters enables a cleaner approach to passing task-specific data, enhancing encapsulation.

Assignment Problem 6

Create an FreeRTOS application.

Create two tasks **Task1** and **Task2,** make use of Task Parameters to pass delay value in Task1 and Task2 at the time of creation from main().

Task1 -> LED 1, 2

Task2 -> LED 3, 3

Example:

If 1000 is passed as parameter at Task1 Creation, Task 1 should blink LEDs at every 1000mS.

If 2000 is passed as parameter at Task2 Creation, Task 1 should blink LEDs at every 2000mS.

Note- Understand the pointer typecasting and usage.

Tip- Try passing the value by refrence.

TASK MANAGEMENT WITH TAKS HANDLES

Task Deletion

- vTaskDelete: Deletes a task specified by its task handle, terminating its execution

Task Suspension and Resumption

- vTaskSuspend: Suspends the execution of a task specified by its task handle.
- vTaskResume: Resumes the execution of a previously suspended task specified by its task handle.

Task Priority Management

- vTaskPrioritySet: Sets the priority of a task specified by its task handle.
- uxTaskPriorityGet: Retrieves the priority of a task specified by its task handle.

Task Notification

- ulTaskNotifyTake: Allows a task to take a notification from another task specified by its task handle.
- xTaskNotify: Sends a notification to a task specified by its task handle.

Task Information Retrieval

- pcTaskGetName: Retrieves the name of a task specified by its task handle.
- eTaskGetState: Retrieves the state of a task specified by its task handle.
- xTaskGetCurrentTaskHandle: Retrieves the task handle of the currently executing task.

TASK Handle

Description

- In FreeRTOS, task handles are used to uniquely identify and manage tasks.
- A task handle in FreeRTOS is a variable of type **TaskHandle_t**.

Creation

- Task handles are typically created when a task is created using the **xTaskCreate** function.
- The handle is assigned a value by the FreeRTOS scheduler, allowing external referencing of the task.

Task Identification

- The task handle serves as a unique identifier for a specific task in the system.
- It allows other tasks or parts of the system to interact with or manage a specific task.

Passing Between Tasks

- Task handles can be passed between tasks or functions to facilitate communication or synchronization between different parts of the system.
- This is often used when one task needs to signal or control the behavior of another task.
- Further controls involves Task Deletion, Task Synchronization, Debugging and Monitoring

TASK Handle

Handles

A task handle is a reference to a specific task. Task handles are used to uniquely identify and manage tasks within the operating system.

Task Management

Allow you to control and manipulate tasks after they have been created. Task handles are typically of type TaskHandle_t, which is a pointer to a task control block (TCB)

Getting the Task Handle

When you create a task using the xTaskCreate or a similar function, you can pass a pointer to a TaskHandle_t variable. This variable will be populated with the task's handle if the task is created successfully.

Example

```
TaskHandle_t xTaskHandle;

xTaskCreate(vTaskFunction, "Task Name", 200, NULL, 2, &xTaskHandle);
```

Assignment Problem 7

Create an FreeRTOS application.

Create Task1 and Task2 with their proper TaskHandles.

Pass the following parameters:

Task 1 (handle) ---Pass to---> Task 1 as parameter

Task 2 (handle) ---Pass to---> Task 2 as parameter

Inside Task1:

Blink LED 1 and LED 2 at every 3 mS for 10 time, after that delete itself.

Inside Task2:

Blink LED 3 and LED 4 at every 5 mS for 5 time, after that delete itself.

Assignment Problem 8

Create an FreeRTOS application.

Create Task1 and Task2 with their proper TaskHandles.

Pass the following parameters:

Task 2 (handle) ---Pass to---> Task 1 as parameter

Task 1 (handle) ---Pass to---> Task 2 as parameter

Inside Task1:

Suspend Task2 using its handle using proper API.

Blink LED 1, 2 at 500mS 10 Times, after that Resume Task 2 using proper API

Inside Task2:

Blink LED 3 and LED 4 at every 100 mS infinite times.

Assignment Problem 9

Create an FreeRTOS application.

Create Task1 **(Priority 2)** and Task2 **(Priority 3)** with their proper TaskHandles.

Task1

-> Set LED 1 Low

-> wait 250 mS

-> Decrement Task1 priority by 2

Task2

-> Set LED 1 High

-> wait 250 mS

-> Increment Task1 priority by 2

Tip: Read about task priority modification related APIs from document.

Observe the behavior and document

Assignment Problem 10

Create Task1 **(Priority 1)** and Task2 **(Priority 1)** with their proper TaskHandles.

Task1

- After 5000 mS delete Task2.
- Delete itself

Task2

- Toggle LED 1, 2, 3, 4 at every 100 mS.

Scenario 2 (Try for Segger Trace Viewer and get trace)

Task1

- After 25 mS delete Task2.
- Delete itself

Task2

- Toggle LED 1, 2, 3, 4 at every 2 mS.

Tip: Read about task management related APIs from document.

Observe the behavior and document.

We Covered

- FreeRTOS Tracing Tool integration
- FreeRTOS task state
- FreeRTOS task state transition
- Task Scheduling
- Task Handle
- Basic Task Management
- Quick Test

QUICK TEST

THANK YOU

Aman Kanwar