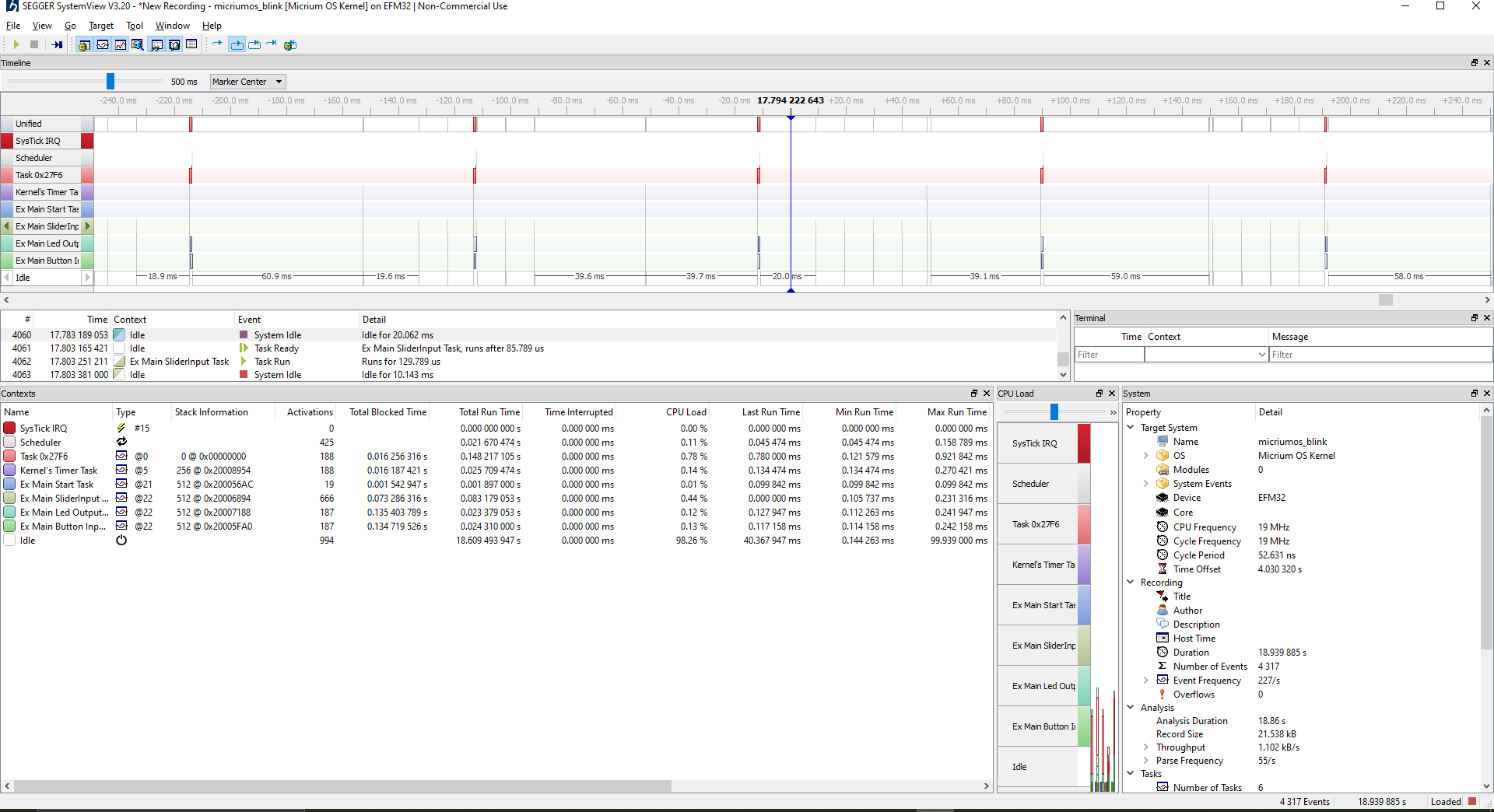
**LAB 3**

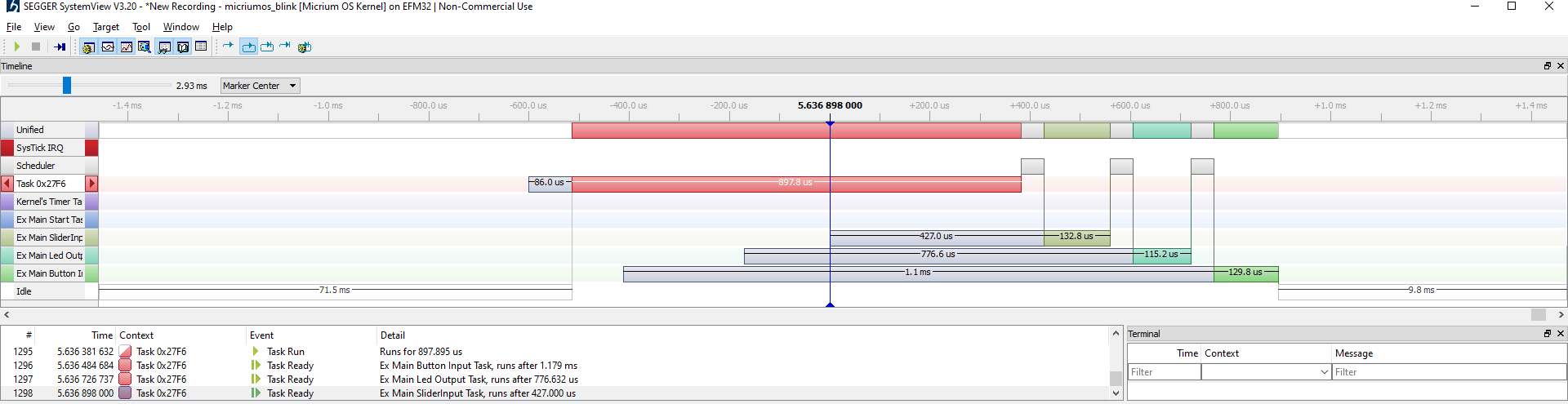
Karston Christensen

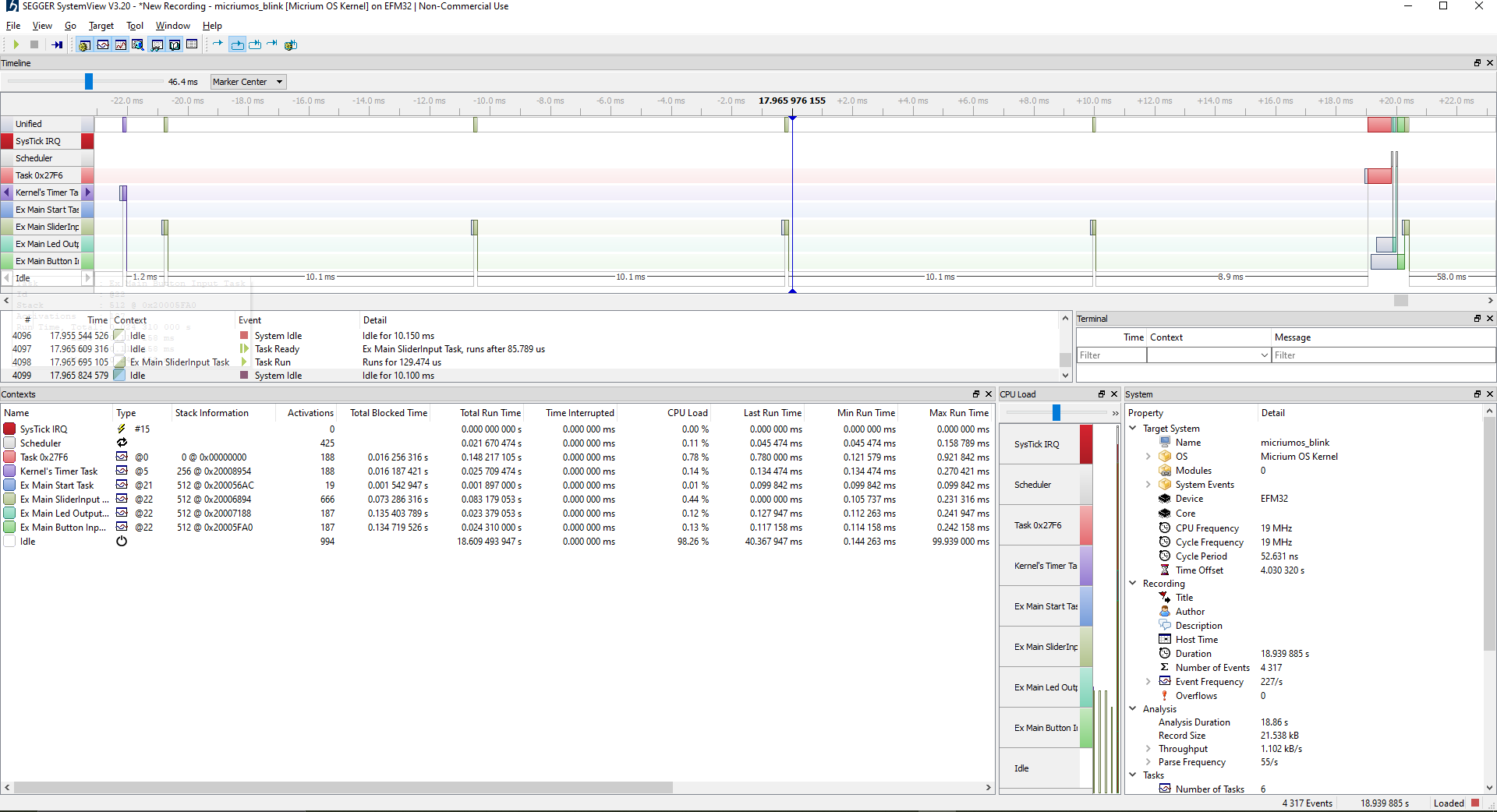
**Lab Questions:**

**13.** The tasks’ start times can be seen in several ways. They can be seen by hovering over the colored task block in the timeline window, by observing the timeline at the top of the timeline window, and by double clicking on the colored task block and reading the corresponding data in the event log.

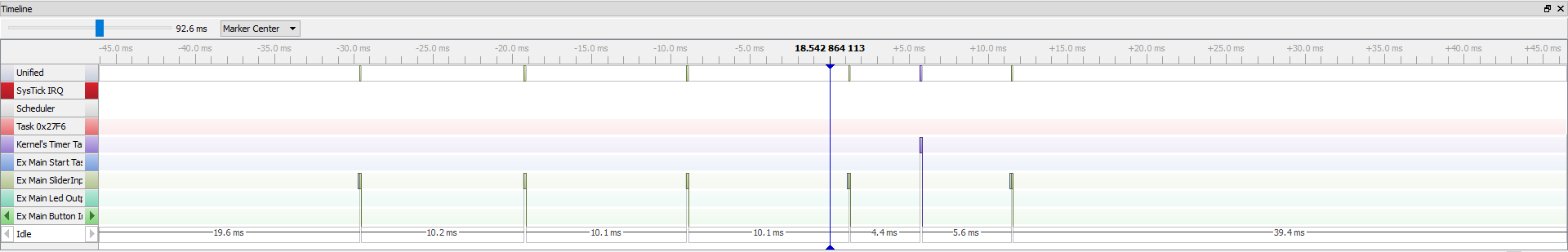


Above is an image of the application window after recording is paused.

A good amount of data can be observed from this window. Above is a visual representation of the timing of tasks by the MicriumOS. Each task’s ready time (while it is waiting in the ready queue) and its run time (while it is using the processor) can be seen. The task’s ready time can be seen in the grey section preceding the task’s execution time, which is shown in color. The ready time can also be seen in the event log, followed by the task’s execution. A “Ready” event in the event log describes when the task entered the ready queue and how long it stayed in the ready queue before running.



Above is an image of the application window with the task timeline window zoomed in to contain a single period of task completion. Many of the tasks that do not have to execute within time constraints are performed in rapid succession of each other, as seen with task 0x27F6, the LED output task, and the button input task. The slider input task must be revisited several times since the measurement of capacitor pads is timing-reliant.

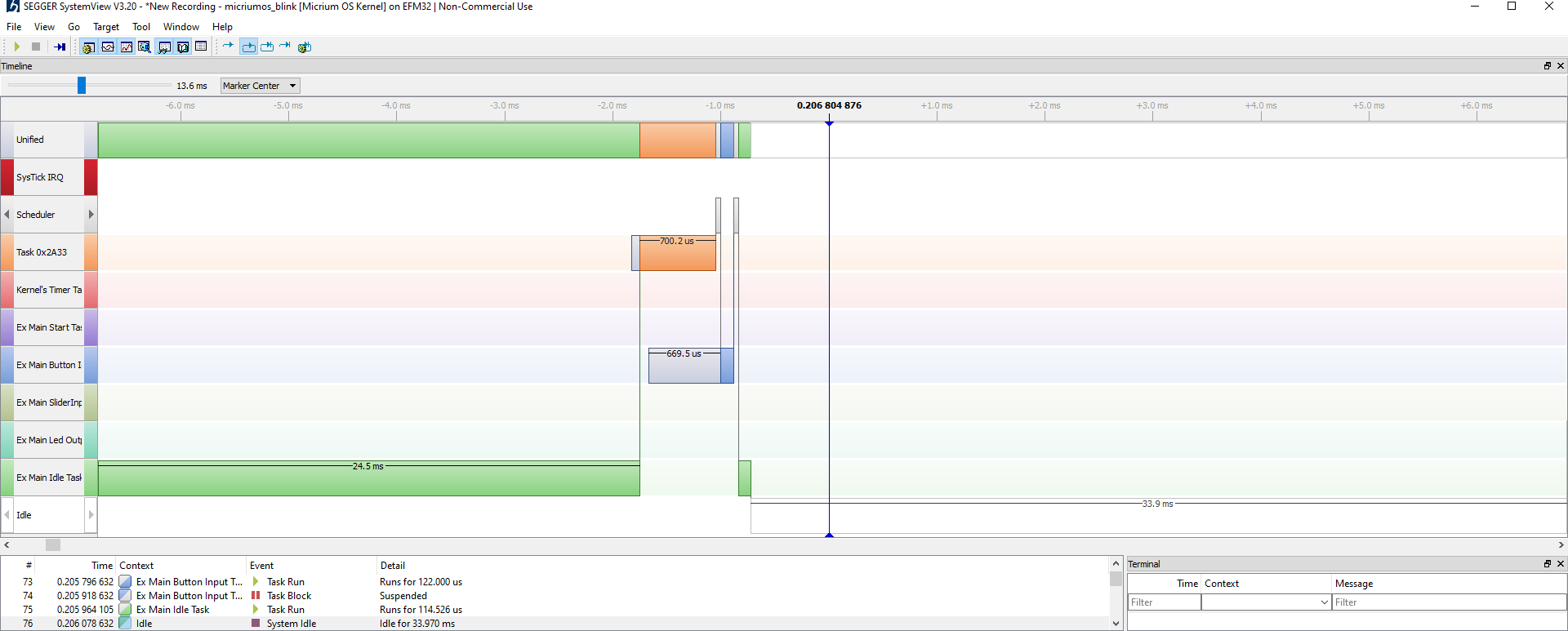


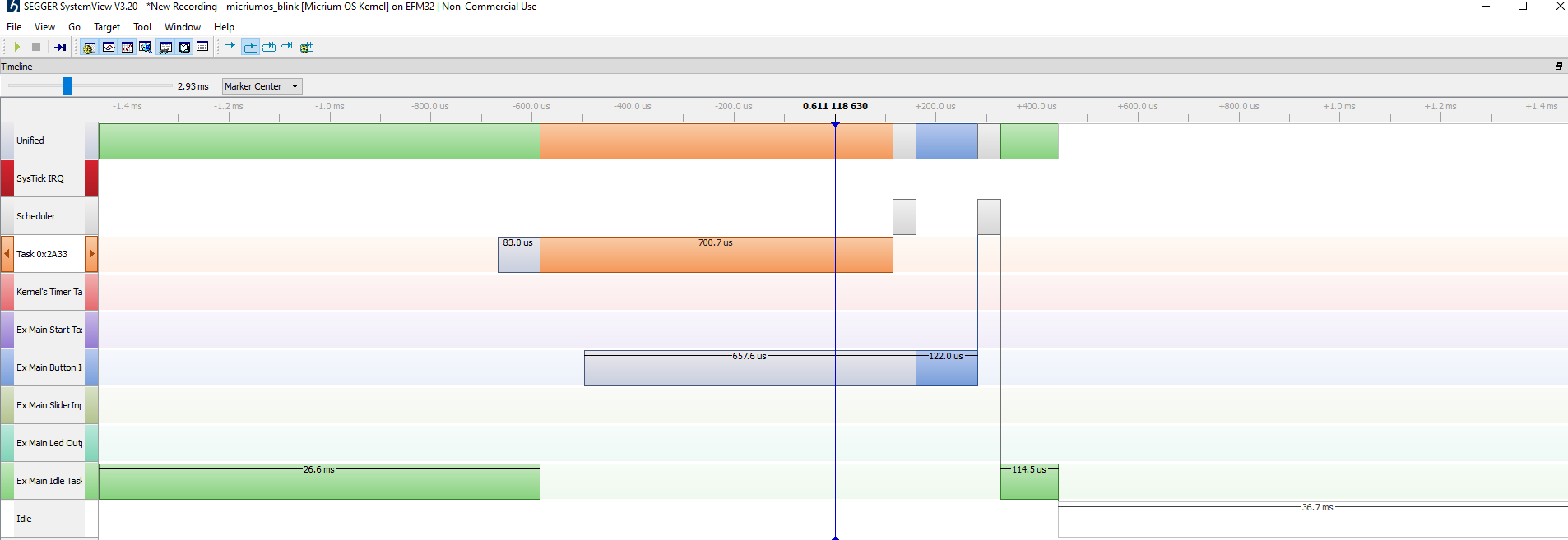
This image displays 5 visits to the Capsense task, each visit occurring within 10 ms of the previous visit. This is exactly as expected since the MicriumOS timer tick increments 10 times between each capacitor pad measurement, providing sufficient time for an RC curve integration to be performed to determine capacitance for the peripheral. My finger typically spans 5 capacitor pads, so this burst of 5 visits to the task is observed as expected.

**14.** The scheduling of the tasks is not what I anticipated it to be. The button input task and the LED output task run in rapid succession of each other every 100 ms. They run within 1 ms of each other. The LED output task runs until it is blocked and a context switch occurs, giving the CPU to the button input task. The button input task then finishes and the context switches to the slider input task.

As mentioned before, when there is an input on the slider, the slider input task is revisited several times throughout the 100 ms period, revisited for approximately 130 microseconds for each capacitor pad that is measured. My finger usually spans 5 to 6 capacitor pads, so the slider input task is revisited 5 to 6 times typically when measurement is occurring. The task visits are spaced out by a span of 10 ms, which is how long I’ve specified for the Capsense module to integrate the RC charging and decay cycle using the Micriumos timer tick to obtain a reading on slider position. This is interesting to see as it’s exactly what I expected to see from the solution proposed by Professor Haines in circumventing the initial incompatibility of the Capsense module and the Micrium OS.

Interrupts claim priority and it is interesting to observe an interrupt routine preempting a normally scheduled task.

Here is an image that displays a context switch. My Idle task (from part 3) is preempted by task 0x27F6. The Main Button Input task is then run after task 0x27F6 yields the processor since it is ready and has a higher priority than the Idle task.



Here is another image of the same context switch with a greater degree of granularity.

The scheduling of the tasks seems optimal to me since the processor has lots of time in between tasks (with the exception of when a slider input measurement must be performed and when interrupts are triggered), which means that it can sleep for a long period before being woken up to attend to quick tasks. The processor usually does not have tasks scheduled for a duration of around 99 ms out of a 100 ms period, which means that it is not often being woken up and put to sleep. Waking up the processor could expend additional energy, so it is preferable for it to sleep for as long as possible.

**15.**

From the window above, the frequency, maximum run time, and minimum run time of each task can be easily observed. The frequencies, maximum run times, and minimum run times of my tasks are documented below:

**Main start task:**

Executed once approximately every second since there is a delay of 1000 ms in the start task.

Frequency: 1 Hz.

Minimum run time: 0.089053

Maximum run time: 0.099842 ms

**Task 0x27F6:**

Frequency: 10 Hz

Minimum run time: 0.045474 ms

Maximum run time: 0.141737 ms

**Kernel’s Timer Task:**

Frequency: 10 Hz

Minimum run time: 0.123684 ms

Maximum run time: 0.134579 ms

**Main slider input task:**

Executed 5 times every 140 ms. Each of the 5 task executions are performed with 10 ms spacing.

Frequency: 7.142 Hz (if every burst of 5 task visits is one cumulative execution).

Minimum run time: 0.105737 ms.

Maximum run time: 0.231316 ms.

**Main LED output task:**

Executed once every 100 ms.

Frequency: 10 Hz.

Minimum run time: 0.112263 ms

Maximum run time: 0.241947 ms

**Main button input task:**

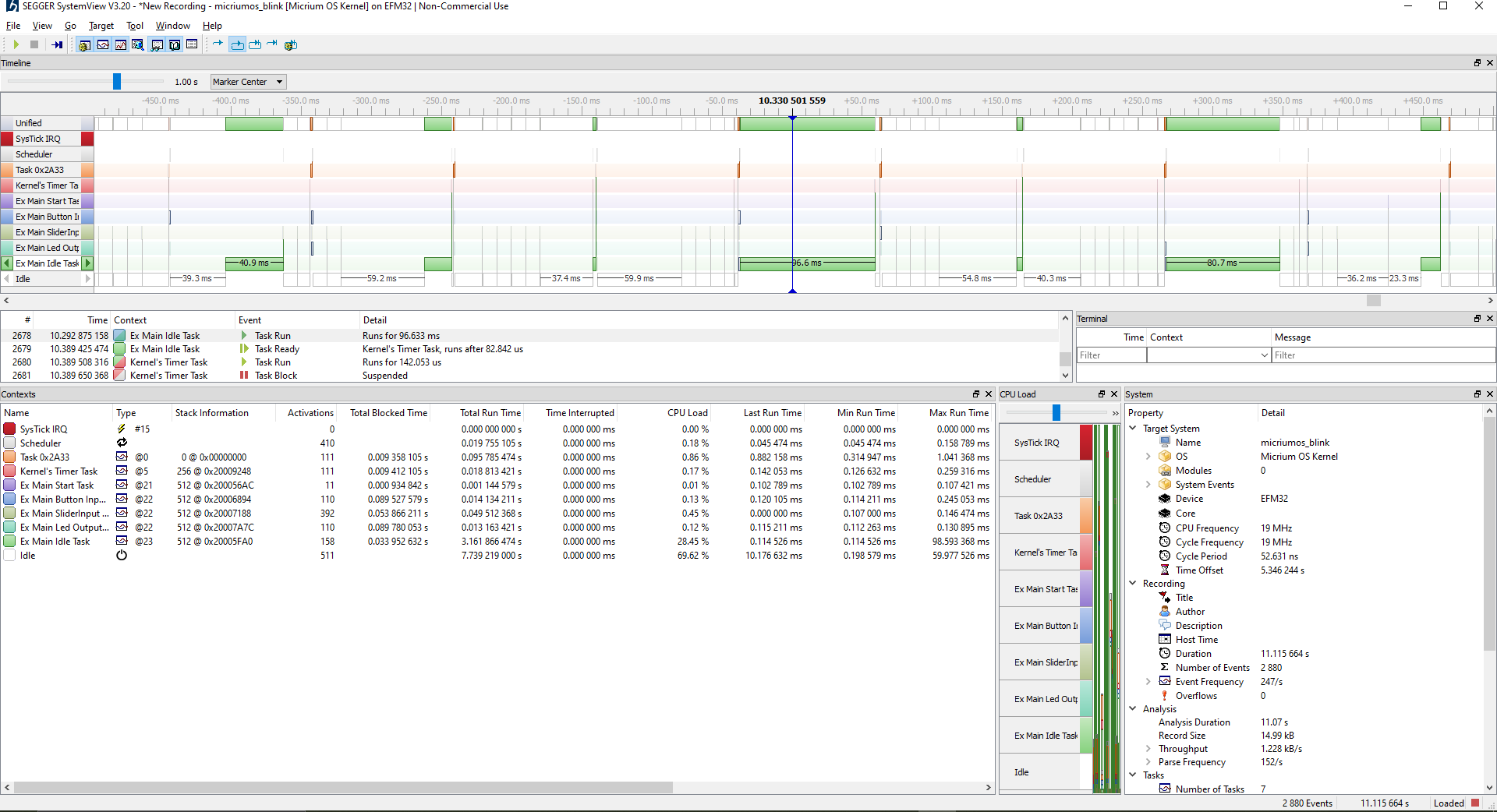
Executed once every 100 ms.

Frequency: 10 Hz.

Minimum run time: 0.114158 ms

Maximum run time: 0.242158 ms

**Part 3: Inclusion of My Own Idle Task**



Above is an image that shows the priority level of the Idle task. The Idle task’s priority is lower than all of the other tasks which is why it is often either preempted by other tasks or postponed from running until all other ready tasks have run.

**1.** The current draw and power draw of the system with the default Idle task provided by MicriumOS are recorded below for the case where no buttons or sliders are pressed.

Average current: 1.49 mA

Average power: 4.95 mW

In Lab 2, these values were recorded for both the IRQ-based implementation and the polling-based implementation. These are as follows:

IRQ Implementation:

Power: 3.33 mW

Polling Implementation:

6.68 mW

**6.** Later, I implemented my own Idle task in which the processor is put into EM1 power level. This yielded the following current draw and power draw in nominal conditions:

Average current: 1.36 mA

Average power: 4.51 mW

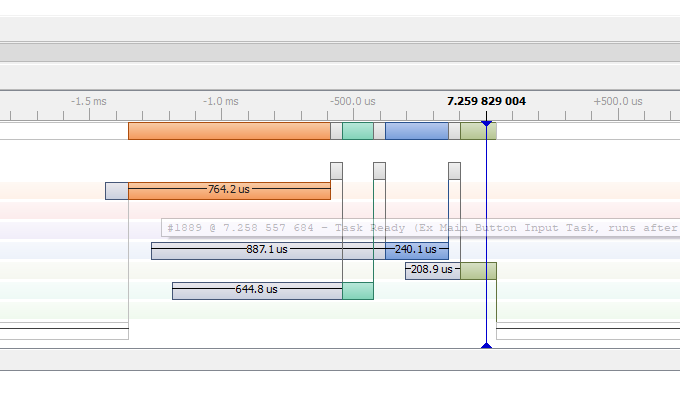
**Comparison:**

The OS implementation wherein the processor is put into EM1 energy mode is more energy-efficient than the original method that uses MicriumOS’s default Idle task. The latter method uses 0.44 mW less on average at nominal conditions. Both MicriumOS solutions are more energy-efficient than the polling method. However, the operating system does consume power and usage of the CPU, so the purely IRQ driven implementation from Lab 2 is still the most energy-efficient, a consuming 1.18 mW less on average than the most efficient MicriumOS solution attempted.

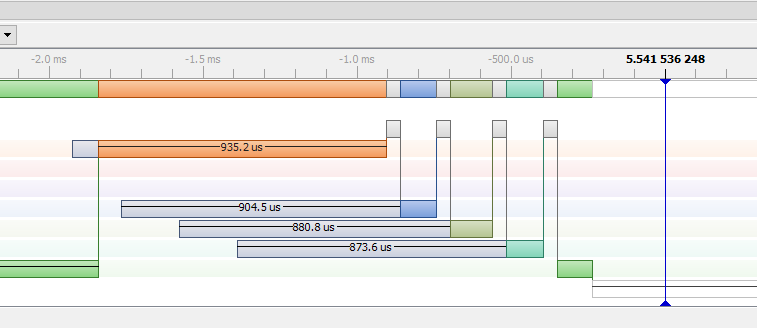
**Bonus Points:**

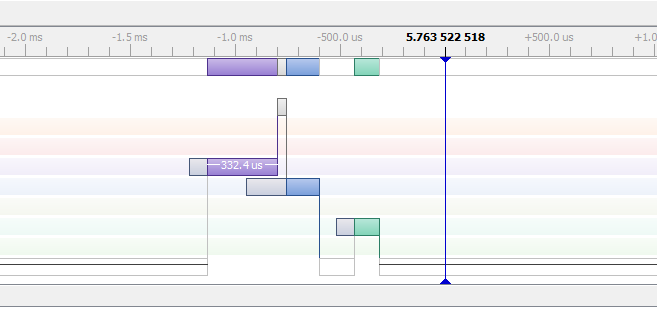
**Changing the priorities of tasks:**

Changing the priorities of the tasks determines in what order they are executed when they are on the ready queue.



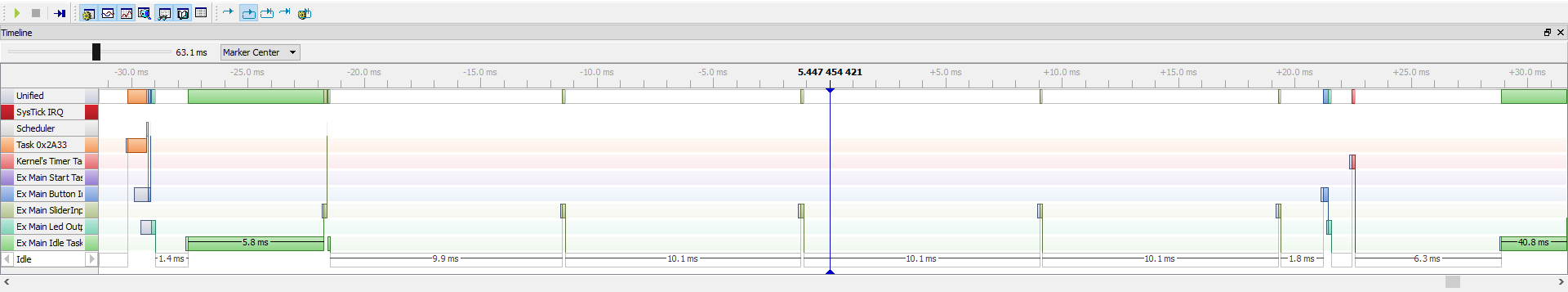
The above image shows a sequence of task executions when the tasks are given equal priority. They are executed in no manner in particular. What is interesting to me is how the blue task was ready before the green task, but the green task was executed before the blue task. Perhaps this has to do with a blocking situation. There is still a considerable time spent in the ready queue for each task as they are executed in rapid succession of one another. This leads to a high wait time.

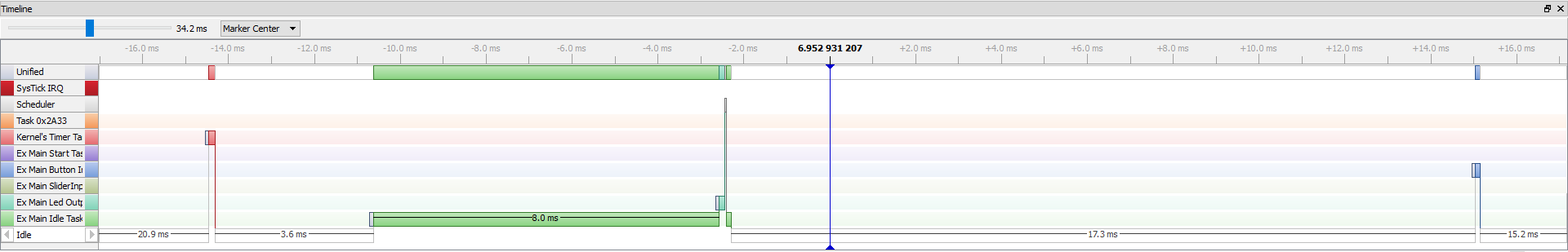
The above image shows the tasks with ascending priority. Here, the button input task has the highest priority of the user defined tasks, followed by slider input, then LED output, and finally Idle. The order of execution of the tasks reflects this prioritization. Again, the tasks spend a considerable amount of time in the ready queue since the tasks are executed right after one another.

Here is an image of the tasks ordered with descending priority. Here the purple Idle task has highest priority, followed by LED output, slider input, and buttons input. The order was hard to capture as the tasks did not always align conveniently due to the additional task visits required for the slider input task, but this image demonstrates the ordering for three of the four user-defined tasks. The problem with wait time remains.

**Changing frequency of tasks:**

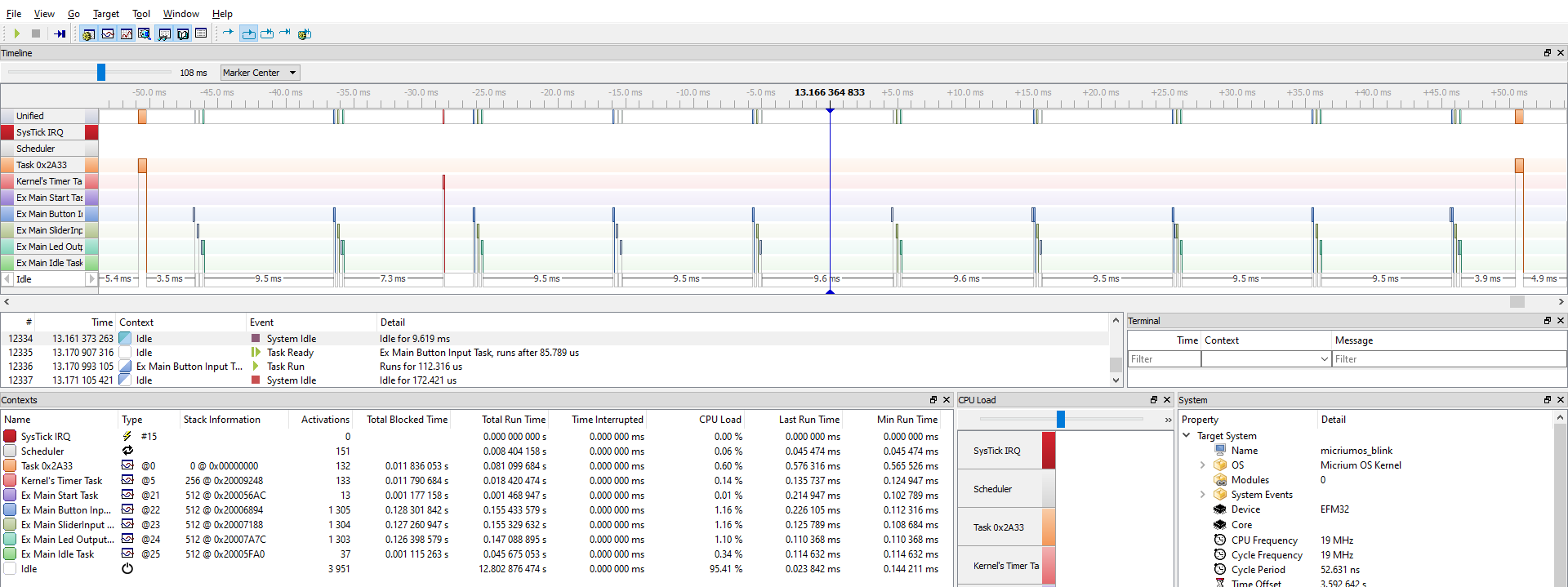
The relative frequency of the tasks can be changed as well. The frequency of a task can be changed by varying the duration of OS timer ticks spent for each OSTimeDly function call in each task function. Here are some images of different frequency combinations for the tasks.

In the above image, the frequency of the tasks is increased from 10 Hz to 50 Hz. The proportion of time wherein the processor is executing task code is much higher, with less sparse space for the processor to sleep in EM1. Since the tasks are all given the same frequency, the wait time issue remains as the tasks are all executed in rapid succession of one another. This problem can be solved by spacing out the tasks so that they are not inserted into the ready queue around the same time.

Above is an image demonstrating spacing of tasks such that they are not added to the ready queue at similar times. Here the tasks are made to wait for dissimilar duration of OS timer ticks, with one task waiting for 50 ticks, one for 60, one for 70, and one for 80. This causes the tasks to spend very little time waiting for other tasks to complete in the ready queue since a task is often the only task waiting to run.

**Achieve less than 10% CPU Load for Idle Task:**

Finally, the CPU usage can be monitored. Although impractical, the frequencies of each task can be modified so that the time spent in the Idle task is very low relative to the time spent on other tasks. This is sub-optimal since time spent in the Idle task is time spent asleep for the processor, saving energy.

In the above image, the other user-defined tasks run with such high frequency, and the idle task with such low frequency relative to the other tasks, that the CPU load for the Idle task is only 0.34%. In this case, each user-defined task is run every 10 OS timer ticks, with the exception of the Idle task, which is run every 1000 OS timer ticks. For comparison, the CPU loads for the button input, slider input, and LED output tasks are all above 1%. Indeed, the power consumption for this situation is much higher than it is for a spaced out task load with high Idle task CPU load. With an Idle task CPU load of 0.34%, the power consumed at nominal conditions is 5.11 mW. That is high compared to the 4.51 mW consumed for a sufficiently spaced out task load.