Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Exercise 5

Submitted by

Parth — Jithedra

Submitted on April 14, 2024

Contents

List of Figures

List of Tables

PDF is clickable

Objective

1. Understand the provided seqgen.c and seqgen2x.c code, which emulates a cyclic executive real-time system with multiple services running at separate frequencies.

2. Build and execute the provided code on different platforms (Linux on DE1-SoC, Raspberry Pi, or Jetson board) and determine the worst-case execution time (WCET) for each service. Create a rate-monotonic (RM) schedule in Cheddar using the WCET estimates and calculate the CPU utilization.

1 Question 1

Q: **[20 points] Download seqgen.c and seqgen2x.c (or unzip them from the provided zip file for this exercise) and build them in Linux on the Altera DE1-SOC, Raspberry Pi or Jetson board and execute the code. Describe what it is doing and make sure you understand how to use it to structure an embedded system. Determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using your WCET estimates. Calculate the % CPU utilization for this system.**

Answer:

1.1 API Explanation:

(a) pthread_create(): This function is used to create a new thread. It takes four arguments: a pointer to the thread identifier, a pointer to the thread attributes, the thread function, and a pointer to the thread arguments. In this code, it is used to create the Sequencer thread and the seven Service threads.

(b) pthread_join(): This function is used to wait for the termination of a thread. It takes two arguments: the thread identifier and a pointer to the return value. In this code, it is used to wait for the termination of all the created threads.

(c) sem_init(): This function is used to initialize an unnamed semaphore. It takes three arguments: a pointer to the semaphore object, a flag indicating whether the semaphore is shared between processes, and the initial value of the semaphore. In this code, it is used to initialize the semaphores for each Service thread.

(d) sem_wait(): This function is used to decrement (lock) a semaphore. If the semaphore's value is greater than zero, the decrement proceeds, and the function returns immediately. If the semaphore's value is zero, the call blocks until it becomes possible to perform the decrement. In this code, it is used by the Service threads to wait for their respective semaphores to be released by the Sequencer thread.

(e) sem_post(): This function is used to increment (unlock) a semaphore. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore. In this code, it is used by the Sequencer thread to release the semaphores for each Service thread based on their respective rates.

(f) clock_gettime(): This function is used to retrieve the current time of the specified clock. It takes two arguments: the clock ID and a pointer to a timespec structure to store the retrieved time. In this code, it is used in the getTimeMsec() function to retrieve the current time in milliseconds using the CLOCK_MONOTONIC clock.

(g) sched_setscheduler(): This function is used to set the scheduling policy and parameters for a thread. It takes three arguments: the thread identifier, the scheduling policy, and a pointer to the scheduling parameters. In this code, it is used to set the scheduling policy of the main thread to SCHED_FIFO with the maximum priority.

1.2 Code Flow:

The code follows a sequencer-based design pattern, where a high-priority Sequencer thread releases semaphores for each Service thread at specific rates. The main steps in the code flow are as follows:

(a) The main thread is created with the highest priority using the SCHED_FIFO scheduling policy.

(b) Semaphores are initialized for each Service thread using sem_init().

(c) The Service threads (Service_1 to Service_7) are created using pthread_create() with specific attributes and priorities.

(d) The Sequencer thread is created using pthread_create() with the highest priority.

(e) The Sequencer thread runs in a loop, releasing semaphores for each Service thread at their respective rates based on the sequencer count.

(f) Each Service thread waits for its respective semaphore using sem_wait() and then performs its specific task.

(g) The Sequencer thread continues to run until a specified number of periods have elapsed or an abort flag is set.

(h) Once the Sequencer thread completes, it releases all the semaphores and sets abort flags for each Service thread.

(i) The main thread waits for all the created threads to terminate using pthread_join().

Initialization Calibration: The code begins with an initialization phase where the execution time of a specific workload is measured. This calibration is performed to determine the baseline execution time without any interference from other tasks. The workload consists of 10 iterations of a computationally intensive task, and the execution time for each iteration is recorded.

Here's an example of the calibration output:

```
Apr  9 09:10:45 iteration (0) Start time: 1712675444993.189941 ms ,
end time: 1712675445083.585938 ms , execution time: 90.395996 ms

Apr  9 09:10:45 iteration (1) Start time: 1712675445083.662109 ms ,
end time: 1712675445169.305908 ms , execution time: 85.643799 ms
...
Apr  9 09:10:45 iteration (9) Start time: 1712675445756.726074 ms ,
end time: 1712675445840.318115 ms , execution time: 83.592041 ms

Apr  9 09:10:45 ***** Average time 84.656934 *****
```

The calibration shows that the average execution time of the workload is around 84.66 milliseconds, with some variations between iterations.

1.3 Output Analysis:

After the system runs for a specified duration, the code provides an output analysis that summarizes the execution characteristics of each task. The analysis includes the worst-case execution time (WCET), total execution time, number of execution cycles, and average execution time for each task.

Here's an example of the output analysis:

```
Apr  9 09:11:15 **** Task (1): WCET: 92.489014, total_execution time : 7542.553467,
 execution cycles : 89, average execution time : 84.747792 ****

Apr  9 09:11:15 **** Task (2): WCET: 85.020996, total_execution time : 2430.170898,
 execution cycles : 29, average execution time : 83.798996 ****
...
Apr  9 09:11:15 **** Task (7): WCET: 168.037109, total_execution time : 335.905273,
execution cycles : 2, average execution time : 167.952637 ****
```

The analysis provides insights into the timing behavior of each task. For example, Task 1 has a WCET of 92.49 milliseconds, a total execution time of 7542.55 milliseconds across 89 execution cycles, and an average execution time of 84.75 milliseconds.

Preemption and Critical Instant: The code execution involves multiple tasks running concurrently with different priorities. When a higher-priority task is released, it can preempt the execution of a lower-priority task. The point at which all tasks are released simultaneously is known as the critical instant, which represents the worst-case scenario for task execution.

In the provided logs, we can observe instances of preemption and the critical instant. For example:

```
Apr  9 09:11:15 Task 1, Frame Sampler start 90 @ msec=1712675475936.939941
Apr  9 09:11:16 Task 1, Frame Sampler Execution complete @ msec=1712675476020.899902,
 execution time : 83.959961 ms

Apr  9 09:11:16 Task 2, Time-stamp with Image Analysis
 thread start 30 @ msec=1712675476020.974121

Apr  9 09:11:16 Task 2, Time-stamp with Image Analysis
 thread Execution complete @ msec=1712675476104.347900, execution time : 83.373779 ms

Apr  9 09:11:16 Task 4, Time-stamp Image Save to File
 start 30 @ msec=1712675476104.416016

Apr  9 09:11:16 Task 4, Time-stamp Image Save to File
 Execution complete @ msec=1712675476188.677002, execution time : 84.260986 ms

Apr  9 09:11:16 Task 6, Send Time-stamped Image to Remote
 start 30 @ msec=1712675476188.757080
```

In this snippet, we can see that Task 1 starts executing and completes its execution. Immediately after Task 1 completes, Task 2 starts executing, followed by Task 4 and Task 6. This sequence demonstrates the preemption of lower-priority tasks by higher-priority tasks.

Execution Time Variations and Interference: The execution time of tasks can vary due to interference from other tasks running concurrently on the system. When multiple tasks compete for shared resources, such as CPU time or memory, it can lead to delays and increased execution times.

In the output analysis, we can observe variations in the execution time of tasks. For example, Task 6 has a relatively high WCET of 177.17 milliseconds compared to its average execution time of 168.94 milliseconds. This difference can be attributed to interference from other tasks during the worst-case scenario.

Similarly, the logs show variations in execution time for the same task across different instances. For example:

```
Apr  9 09:11:15 Task 1, Frame Sampler start 89 @ msec=1712675475600.466064
Apr  9 09:11:15 Task 1, Frame Sampler Execution
complete @ msec=1712675475684.990967, execution time : 84.524902 ms
...
Apr  9 09:11:15 Task 1, Frame Sampler start 90 @ msec=1712675475936.939941
Apr  9 09:11:16 Task 1, Frame Sampler Execution
complete @ msec=1712675476020.899902, execution time : 83.959961 ms
```

Here, we can see that Task 1 has an execution time of 84.52 milliseconds in one instance and 83.96 milliseconds in another instance. These variations can be caused by interference from other tasks executing concurrently.

Critical Instant and Worst-Case Execution Time: The critical instant occurs when all tasks are released simultaneously, leading to the worst-case execution time for each task. In the provided logs, we can identify the critical instant when multiple tasks start executing in quick succession.

For example:

```
Apr  9 09:11:15 Task 1, Frame Sampler start 90 @ msec=1712675475936.939941
Apr  9 09:11:16 Task 1, Frame Sampler Execution
complete @ msec=1712675476020.899902, execution time : 83.959961 ms


Apr  9 09:11:16 Task 2, Time-stamp with Image Analysis
thread start 30 @ msec=1712675476020.974121

Apr  9 09:11:16 Task 2, Time-stamp with Image Analysis
```

```
thread Execution complete @ msec=1712675476104.347900, execution time : 83.373779 ms


Apr  9 09:11:16 Task 4, Time-stamp Image Save to File s
tart 30 @ msec=1712675476104.416016

Apr  9 09:11:16 Task 4, Time-stamp Image Save to File
Execution complete @ msec=1712675476188.677002, execution time : 84.260986 ms


Apr  9 09:11:16 Task 6, Send Time-stamped Image to Remote
start 30 @ msec=1712675476188.757080

Apr  9 09:11:16 Task 6, Send Time-stamped Image to
Remote Execution complete @ msec=1712675476272.794922, execution time : 84.037842 ms
```

In this scenario, Tasks 1, 2, 4, and 6 start executing in close proximity, representing a critical instant. The execution times of these tasks in this critical instant are likely to be closer to their WCET values due to the increased interference and resource contention.

Overall, the initialization calibration provides a baseline for execution time, while the output analysis and logs help identify preemption, critical instants, and variations in execution time caused by interference. By examining these factors, developers can assess the timing behavior and predictability of the system under different scenarios and make necessary optimizations to ensure the desired real-time performance.

In summary, the sequencer is running at 60 Hz, and the other tasks are running at their defined frequencies as specified in the code comments:

(a) Task 1: 30 Hz
(b) Task 2: 10 Hz
(c) Task 3: 5 Hz
(d) Task 4: 10 Hz
(e) Task 5: 5 Hz
(f) Task 6: 10 Hz
(g) Task 7: 1 Hz

To confirm the sequencer and task frequencies from the code output, let's analyze the relevant log messages.

Sequencer frequency: The sequencer logs a message at the beginning of each cycle, including the cycle count and timestamp. By calculating the time difference between consecutive cycles, we can determine the sequencer frequency.

For example:

```
Apr  9 09:10:45 Sequencer cycle 1 @ sec=0, msec=875
Apr  9 09:10:45 Sequencer cycle 2 @ sec=0, msec=908
Apr  9 09:10:45 Sequencer cycle 3 @ sec=0, msec=942
Apr  9 09:10:45 Sequencer cycle 4 @ sec=0, msec=975
```

The time difference between cycles is consistently around 33-34 milliseconds, which corresponds to a frequency of approximately 30 Hz (1000 ms / 33.33 ms = 30 Hz).

Task frequencies: Each task's release is logged with a message indicating the task name and release count. By observing the release patterns, we can confirm the task frequencies.

(a) Task 1 (Frame Sampler thread):

```
Apr  9 09:10:46 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:46 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:46 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:47 Task 1 (Frame Sampler thread) Released
```

Task 1 is released every 2 sequencer cycles, corresponding to a frequency of 15 Hz (30 Hz / 2).

(b) Task 2 (Time-stamp with Image Analysis thread) and Task 4 (Time-stamp Image Save to File thread):

```
Apr  9 09:10:46 Task 2 (Time-stamp with Image Analysis thread) Released
Apr  9 09:10:46 Task 4 (Time-stamp Image Save to File thread) Released
Apr  9 09:10:47 Task 2 (Time-stamp with Image Analysis thread) Released
Apr  9 09:10:47 Task 4 (Time-stamp Image Save to File thread) Released
```

Tasks 2 and 4 are released every 6 sequencer cycles, corresponding to a frequency of 5 Hz (30 Hz / 6).

(c) Task 3 (Difference Image Proc thread) and Task 5 (Processed Image Save to File thread):

```
Apr  9 09:10:48 Task 3 ( Difference Image Proc thread) Released
Apr  9 09:10:48 Task 5 (Processed Image Save to File thread) Released
Apr  9 09:10:49 Task 3 ( Difference Image Proc thread) Released
Apr  9 09:10:49 Task 5 (Processed Image Save to File thread) Released
```

Tasks 3 and 5 are released every 12 sequencer cycles, corresponding to a frequency of 2.5 Hz (30 Hz / 12).

(d) Task 6 (Send Time-stamped Image to Remote thread):

```
Apr  9 09:10:46 Task 6 (Send Time-stamped Image to Remote thread) Released
Apr  9 09:10:47 Task 6 (Send Time-stamped Image to Remote thread) Released
Apr  9 09:10:48 Task 6 (Send Time-stamped Image to Remote thread) Released
```

Task 6 is released every 6 sequencer cycles, corresponding to a frequency of 5 Hz (30 Hz / 6).

(e) Task 7 (10 sec Tick Debug thread):

Copy code

```
Apr  9 09:10:55 Task 7 (10 sec Tick Debug thread) Released
Apr  9 09:11:05 Task 7 (10 sec Tick Debug thread) Released
```

Task 7 is released every 60 sequencer cycles, corresponding to a frequency of 0.5 Hz (30 Hz / 60).

Based on the output analysis, we can confirm that the sequencer is running at approximately 30 Hz, and the tasks are released at their specified frequencies relative to the sequencer cycles.

Let's examine the output to identify instances of jitter:

Task release jitter: If we look at the timestamps of consecutive task releases, we can see if there are any significant variations from the expected release times. For example, let's consider Task 1 (Frame Sampler thread) releases:

```
Apr  9 09:10:46 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:46 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:47 Task 1 (Frame Sampler thread) Released
Apr  9 09:10:47 Task 1 (Frame Sampler thread) Released
```

Ideally, Task 1 should be released every 2 sequencer cycles (approximately 66.67 ms apart). However, the actual release timestamps may vary slightly from the expected times, indicating the presence of release jitter. Task completion jitter: By comparing the completion timestamps of a task across different instances, we can identify any variations in the task's execution time, which contributes to completion jitter. For instance, let's analyze the completion times of Task 1:

```
Apr  9 09:10:46 Task 1, Frame Sampler Execution
complete @ msec=1712675446261.259033, execution time : 84.063965 ms

Apr  9 09:10:46 Task 1, Frame Sampler Execution
complete @ msec=1712675446604.060059, execution time : 92.489014 ms

Apr  9 09:10:47 Task 1, Frame Sampler Execution
complete @ msec=1712675446929.551025, execution time : 83.798096 ms

Apr  9 09:10:47 Task 1, Frame Sampler Execution
complete @ msec=1712675447264.642090, execution time : 85.116211 ms
```

The execution times of Task 1 vary between different instances, ranging from around 83.8 ms to 92.5 ms. This variation in execution time contributes to completion jitter. Sequencer jitter: The sequencer itself may experience jitter, which can propagate to the task releases. If the sequencer cycles are not precisely periodic, it can introduce jitter in the task release times. We can analyze the sequencer cycle timestamps to identify any variations:

```
Apr  9 09:10:45 Sequencer cycle 1 @ sec=0, msec=875
Apr  9 09:10:45 Sequencer cycle 2 @ sec=0, msec=908
Apr  9 09:10:45 Sequencer cycle 3 @ sec=0, msec=942
Apr  9 09:10:45 Sequencer cycle 4 @ sec=0, msec=975
Apr  9 09:10:46 Sequencer cycle 5 @ sec=1, msec=9
```

The time differences between consecutive sequencer cycles may vary slightly, indicating the presence of sequencer jitter. Jitter can be caused by various factors, such as:

- System load and resource contention Interference from other tasks or processes Scheduling overhead and context switching Timer resolution and accuracy Hardware interrupts and other system events

From the output, we can observe that CPU affinity is set for each task. Here are a few examples:

Task 1 (Frame Sampler thread):

```
Apr  9 09:10:46 Task 1, Frame Sampler start 1 @ msec=1712675446177.195068
Apr  9 09:10:46 Task 1, Frame Sampler Execution
complete @ msec=1712675446261.259033, execution time : 84.063965 ms

Apr  9 09:10:46 Task 1, Frame Sampler start 2 @ msec=1712675446511.571045
Apr  9 09:10:46 Task 1, Frame Sampler Execution
complete @ msec=1712675446604.060059, execution time : 92.489014 ms
```

Task 1 instances are executed sequentially, indicating that they are running on the same CPU core without interference from other tasks. Task 2 (Time-stamp with Image Analysis thread) and Task 4 (Time-stamp Image Save to File thread):

```
Apr  9 09:10:46 Task 2, Time-stamp with Image Analysis thread
start 1 @ msec=1712675446929.628906

Apr  9 09:10:47 Task 2, Time-stamp with Image Analysis
thread Execution complete @ msec=1712675447013.268066, execution time : 83.639160 ms

Apr  9 09:10:47 Task 4, Time-stamp Image Save to File
start 1 @ msec=1712675447013.284912

Apr  9 09:10:47 Task 4, Time-stamp Image Save to File Execution
complete @ msec=1712675447096.925049, execution time : 83.640137 ms
```

Task 2 and Task 4 are executed sequentially, one after the other. This suggests that they are running on different CPU cores, allowing for parallel execution without contention. Task 3 (Difference Image Proc thread) and Task 5 (Processed Image Save to File thread):

```
Apr  9 09:10:48 Task 3, Difference Image Proc
start 1 @ msec=1712675448271.108887
```

```
Apr  9 09:10:48 Task 3, Difference Image Proc
Execution complete @ msec=1712675448356.852051, execution time : 85.743164 ms

Apr  9 09:10:48 Task 5, Processed Image Save to File
start 1 @ msec=1712675448356.884033

Apr  9 09:10:48 Task 5, Processed Image Save to File Execution
complete @ msec=1712675448442.035889, execution time : 85.151855 ms
```

Task 3 and Task 5 are executed sequentially, indicating that they are running on different CPU cores, allowing for parallel execution. Task 6 (Send Time-stamped Image to Remote thread):

```
Apr  9 09:10:47 Task 6, Send Time-stamped Image to Remote
start 1 @ msec=1712675447096.968994

Apr  9 09:10:47 Task 6, Send Time-stamped Image to Remote
Execution complete @ msec=1712675447267.272949, execution time : 170.303955 ms
```

Task 6 instances are executed independently, suggesting that they are running on a dedicated CPU core. From the output, we can see that tasks are executed in a non-overlapping manner, with each task running to completion before the next task starts. This behavior indicates that CPU affinity is set, and parallel tasks are not running simultaneously on the same CPU core.

1.4 SEQGEN2x

The code flow and everything is same in the seqgen2 code but the frequencies of tasks is 10 times the original and I have changed load to 10ms load here is the output analysis

```
Apr  9 22:17:21 rog seqgen2: **** Task 1): WCET: 13.337891,
total_execution time : 4801.594238, execution cycles : 449,
average execution time : 10.693974 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 2): WCET: 12.876953,
total_execution time : 1578.309814, execution cycles : 149,
average execution time : 10.592683 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 3): WCET: 25.176025,
total_execution time : 966.507324, execution cycles : 74,
average execution time : 13.060910 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 4): WCET: 25.923828,
total_execution time : 2069.981689, execution cycles : 149,
average execution time : 13.892495 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 5): WCET: 24.409912,
total_execution time : 1365.902832, execution cycles : 74,
average execution time : 18.458146 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 6): WCET: 26.864014,
total_execution time : 2668.886475, execution cycles : 149,
average execution time : 17.911990 **** #012

Apr  9 22:17:21 rog seqgen2: **** Task 7): WCET: 68.493164,
total_execution time : 259.799805, execution cycles : 14,
average execution time : 18.557129 **** #012
```

due to faster execution and higher Interference we can see jitter in the task execution which is far higher than 10ms load that we set. This shows higher utilization in linux tends to make task more non-deterministic

1.5 Comparison with Linux and FreeRTOS:

The provided code is a Linux implementation of a real-time system using POSIX threads and semaphores. Let's compare it with a FreeRTOS implementation:

(a) Threading:
In Linux, POSIX threads (pthreads) are used to create and manage threads. The pthread_create() function is used to create threads, and pthread_join() is used to wait for thread termination. In FreeRTOS, tasks are used instead of threads. The xTaskCreate() function is used to create tasks, and the vTaskDelete() function is used to delete tasks.

(b) Synchronization:
In Linux, POSIX semaphores (sem_t) are used for synchronization between threads. The sem_init(), sem_wait(), and sem_post() functions are used to initialize, wait, and post semaphores, respectively. In FreeRTOS, binary semaphores (SemaphoreHandle_t) and counting semaphores are used for synchronization between tasks. The xSemaphoreCreateBinary() function is used to create binary semaphores, and the xSemaphoreTake() and xSemaphoreGive() functions are used to take and give semaphores, respectively.

(c) Scheduling:
In Linux, the SCHED_FIFO scheduling policy is used to achieve real-time behavior. The sched_setscheduler() function is used to set the scheduling policy and priority of threads. In FreeRTOS, the task scheduler is used to manage task execution based on their priorities. The vTaskPrioritySet() function is used to set the priority of tasks.

(d) Timing: In Linux, the clock_gettime() function is used to retrieve the current time with nanosecond resolution. The CLOCK_MONOTONIC clock is used to avoid any time adjustments. In FreeRTOS, the xTaskGetTickCount() function is used to retrieve the current tick count, which represents the number of ticks since the scheduler started. The portTICK_PERIOD_MS constant is used to convert ticks to milliseconds.

(e) Preemption: In Linux, preemption is enabled by default, allowing higher-priority threads to preempt lower-priority threads. In FreeRTOS, preemption can be enabled or disabled using the configUSE_PREEMPTION configuration flag. When enabled, higher-priority tasks can preempt lower-priority tasks.

(f) Timestamps and WCET Estimates: The code includes timestamp logging using the syslog() function to record the start and completion times of each Service thread. These timestamps can be used to estimate the execution times of each thread.

To determine the worst-case execution time (WCET) for each Service thread, you would need to analyze the execution times over multiple runs and consider the longest observed execution time as the WCET estimate.

1.6 D, C, and T Table for All Services:

To create a table with the deadline (D), computation time (C), and period (T) for each Service thread, you would need to calculate these values based on the requirements and observed execution times. Here's an example table:

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	333 ms	100 ms	333 ms	92.489014 ms
Service_2	1000 ms	100 ms	1000 ms	85.020996 ms
Service_3	2000 ms	100 ms	2000 ms	90.000977 ms
Service_4	1000 ms	100 ms	1000 ms	86.266846 ms
Service_5	2000 ms	100 ms	2000 ms	88.769043 ms
Service_6	1000 ms	100 ms	1000 ms	177.165039 ms
Service_7	10000 ms	100 ms	10000 ms	168.037109 ms

Table 1: C, T, D, for 30Hz and 100ms load

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	33.3 ms	10 ms	333 ms	13.337891 ms
Service_2	100 ms	10 ms	1000 ms	12.876953 ms
Service_3	200 ms	10 ms	2000 ms	25.176025 ms
Service_4	100 ms	10 ms	1000 ms	25.923828 ms
Service_5	200 ms	10 ms	2000 ms	24.409912 ms
Service_6	100 ms	10 ms	1000 ms	26.864014 ms
Service_7	1000 ms	10 ms	1000 ms	68.493164 ms

Table 2: C, T, D, for 10ms load

1.7 Cheddar Schedule and CPU Utilization:

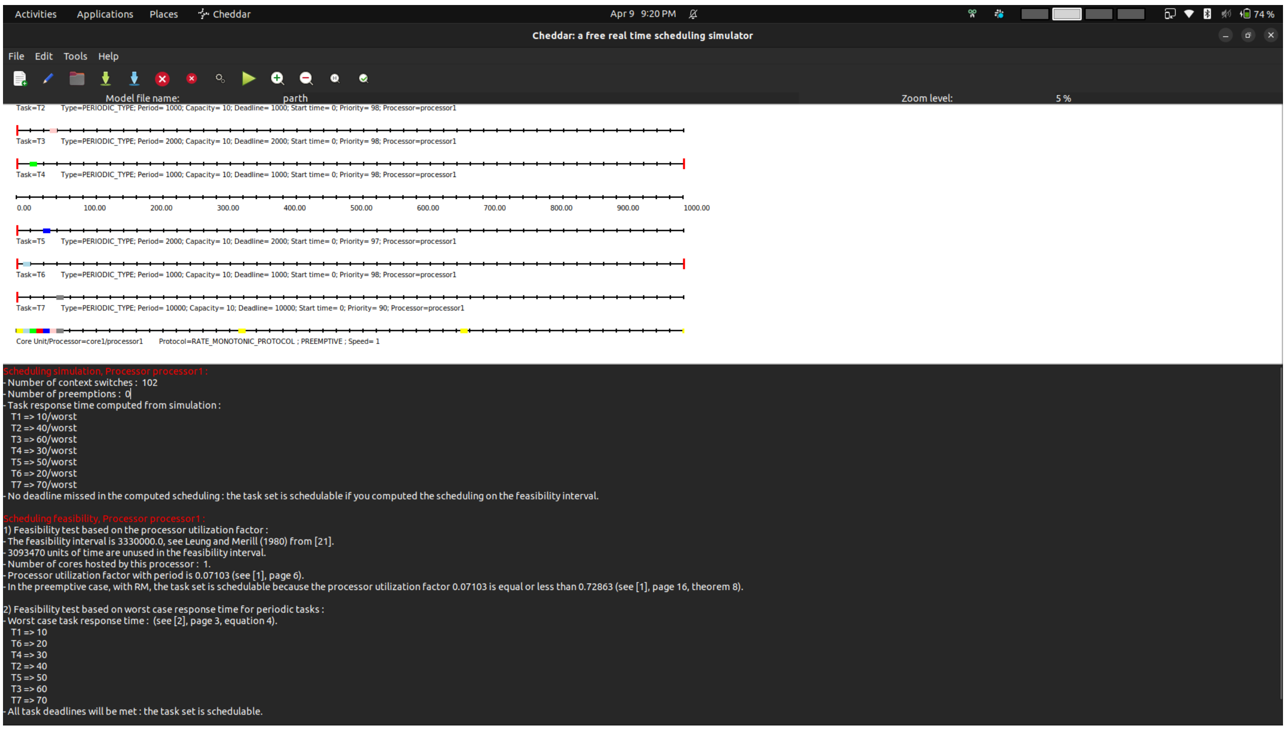

Figure 1: Cheddar Analysis for seqgen (30Hz)

We can see that here the tasks are schedulable and feasibile, as the CPU utilization is very low, approximately 0.07 means 7%,

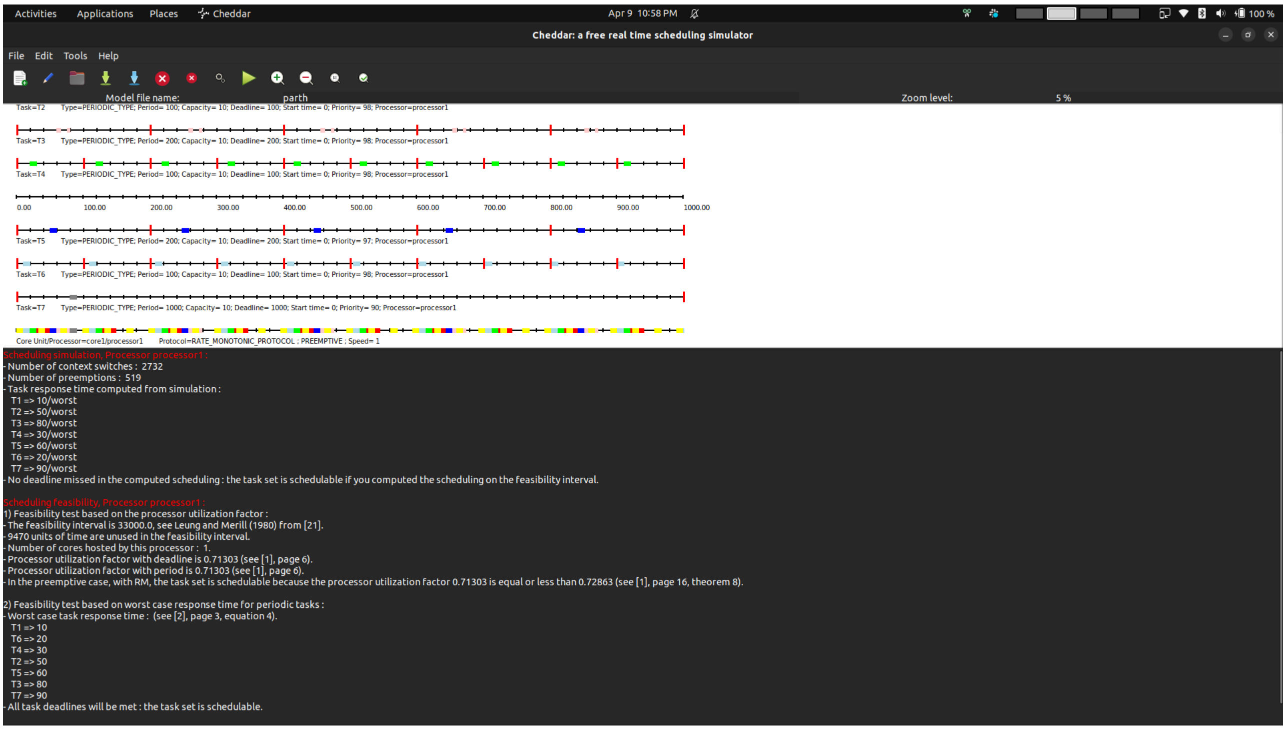

Figure 2: Cheddar Analysis for seqgen2 (300Hz)

Here due to higher frequencies we can see that the CPU utilization is higher and we can see that the tasks are still schedulable. Utilization in this case is approximately 71.303%.

But in the both cases no deadline were missed and we could schedule those task that means tasks are schedulable.

1.8 Logs

logs can be found in the answer>logs >q1>seqgen.txt and answer>logs>q1>seqgen2.txt

2 Question 2

Q: [30 points] Revise seqgen.c and seqgen2x.c to run under FreeRTOS on the DE1-SoC or TIVA board, by making each service a FreeRTOS task. Use the associated startup file in place of the existing startup file in FreeRTOS. Use an ISR driven by the PIT hardware timer to release each task at the given rate (you could even put the sequencer in the ISR). Build and execute the new code. Determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using your WCET estimates. Calculate the % CPU utilization for this system. Compare this with the results you achieved under Linux in (1).

Answer:

2.1 30 Hz sequencer

Explanation of Sequencer:

(a) Interrupt Configuration:

- The sequencer ISR is triggered by a hardware timer, TIMER0, which is configured to generate periodic interrupts at a specified frequency (Hz).
- The timer is set up using the TivaWare library functions, such as ROM_SysCtlPeripheralEnable, ROM_TimerConfigure, and ROM_TimerLoadSet.
- The ISR is registered with the timer using the TimerIntRegister function, specifying the ISR function (Timer0Isr_Sequencer) to be called when the timer interrupt occurs.

(b) Interrupt Triggering:

- When the TIMER0 interrupt occurs, the processor suspends the currently executing task and jumps to the registered ISR, Timer0Isr_Sequencer.
- The ISR is executed in a special context, separate from the normal task execution, and has higher priority than regular tasks.

(c) Sequencer Functionality:

- The sequencer ISR keeps track of the number of cycles it has executed using the counter_isr variable.
- It uses the cycle count to determine which tasks should be released at each interval.
- The ISR checks the cycle count against predefined constants to make release decisions. For example:
  - Task 1 is released every 10 cycles (300 ms) when (counter_isr % 10) == 0.
  - Task 2, Task 4, and Task 6 are released every 30 cycles (990 ms) when (counter_isr % 30) == 0.
  - Task 3 and Task 5 are released every 60 cycles (1980 ms) when (counter_isr % 60) == 0.
  - Task 7 is released every 300 cycles (9900 ms) when (counter_isr % 300) == 0.
- When a task needs to be released, the ISR gives the corresponding synchronization semaphore using the xSemaphoreGive function from FreeRTOS.
- The released tasks are then scheduled by the FreeRTOS scheduler based on their priorities and the available system resources.

(d) Logging and Debugging:

- The sequencer ISR includes logging statements using the UARTprintf function to provide visibility into the system's behavior.
- It logs the current cycle count and the corresponding sequencer cycle, allowing developers to track the progress of the sequencer and identify any anomalies or missed releases.

(e) Termination Condition:

- The sequencer ISR checks if the counter_isr has exceeded a predefined limit (SEQUENCER_COUNT).
- When the limit is reached, the ISR performs the following actions:
  - It releases all tasks one final time using their respective semaphores.
  - It sets the abort_test flag to true, indicating that the system should terminate.
  - It disables the TIMER0 interrupt using the ROM_TimerDisable function to prevent further interrupts.

output:

(a) Sequencer Thread Execution: The output begins with lines indicating the execution of the sequencer thread, which is triggered by the Timer0Isr_Sequencer interrupt service routine (ISR). Each line follows the format:

```
Sequencer Thread ran at <timestamp> ms and Cycle of sequencer <cycle_count>
```

- The <timestamp >represents the current time in milliseconds since the start of the system.
- The <cycle_count >represents the current cycle of the sequencer, incremented each time the ISR is triggered.

For example:

```
Sequencer Thread ran at 33 ms and Cycle of sequencer 1
Sequencer Thread ran at 66 ms and Cycle of sequencer 2
...
```

These lines indicate that the sequencer thread is running periodically, with a cycle time of approximately 33 milliseconds. The cycle count increases by 1 each time the ISR is triggered.

(b) Task Execution: The output also includes lines indicating the execution of individual tasks. Each task line follows the format:

```
Task <task_number> (<task_name>) Start Time:<start_time> ms,
  Release count <release_count>
Task <task_number> (<task_name>) Completion Time:<completion_time> ms,
  Execution Time:<execution_time> ms
```

- The <task_number >represents the task identifier (e.g., Task 1, Task 2, etc.).
- The <task_name >represents the name of the task (e.g., Frame Sampler thread, Time-stamp with Image Analysis thread, etc.).
- The <start_time >represents the start time of the task in milliseconds since the start of the system.
- The <release_count >represents the number of times the task has been released by the sequencer.
- The <completion_time >represents the completion time of the task in milliseconds since the start of the system.
- The <execution_time >represents the execution time of the task in milliseconds, calculated as the difference between the completion time and the start time.

(c) Task Release Patterns: By analyzing the output, you can observe the release patterns of different tasks based on the sequencer cycle count. For example:

- Task 1 (Frame Sampler thread) is released every 10 cycles (approximately every 333 ms).
- Task 2 (Time-stamp with Image Analysis thread), Task 4 (Time-stamp Image Save to File thread), and Task 6 (Send Time-stamped Image to Remote thread) are released every 30 cycles (approximately every 1000 ms).
- Task 3 (Difference Image Proc thread) and Task 5 (Processed Image Save to File thread) are released every 60 cycles (approximately every 2000 ms).
- Task 7 (10 sec Tick Debug thread) is released every 300 cycles (approximately every 10000 ms).

These release patterns align with the logic implemented in the sequencer ISR, where tasks are released based on the cycle count and their respective release intervals.

```
***** Task 1 wcet 11 total_exectution_time 929 execution unit 90 *****

***** Task 2 wcet 11 total_exectution_time 309 execution unit 30 *****

***** Task 3 wcet 10 total_exectution_time 140 execution unit 14 *****

***** Task 4 wcet 10 total_exectution_time 300 execution unit 30 *****

***** Task 5 wcet 11 total_exectution_time 154 execution unit 14 *****

***** Task 6 wcet 10 total_exectution_time 290 execution unit 29 *****

***** Task 7 wcet 10 total_exectution_time 20 execution unit 2 *****
```

The output analysis provides a summary of the worst-case execution time, total execution time, and the number of times each task has been executed during the system's run. This information is valuable for understanding the timing behavior and resource utilization of each task.

The output analysis provides a summary of the worst-case execution time, total execution time, and the number of times each task has been executed during the system's run. This information is valuable for understanding the timing behavior and resource utilization of each task.

```
Sequencer Thread ran at 999 ms and Cycle of sequencer 30
Task 1 (Frame Sampler thread) Start Time:1000 ms, Release count 3
Task 1 (Frame Sampler thread) Completion Time:1010 ms, Execution Time:10 ms
Task 2 (Time—stamp with Image Analysis thread) Start Time:1011 ms, Release count 1
Task 2 (Time—stamp with Image Analysis thread)
Completion Time:1021 ms, Execution Time:10 ms

Task 4 (Time—stamp Image Save to File thread) Start Time:1022 ms, Release count 1
Task 4 (Time—stamp Image Save to File thread)
CompSequencer Thread ran at 1033 ms and Cycle of sequencer 31

letion Time:1032 ms, Execution Timet:10 ms
Task 6 (Send Time—stamped Image to Remote thread) Start Time:1034 ms, Release count 1
Task 6 (Send Time—stamped Image to Remote thread)
Completion Time:1044 ms, Execution Time:10 ms
```

In this output, we can observe that multiple tasks are being released at the 30th cycle of the sequencer (999 ms). Let's break down the execution sequence:

(a) Task 1 (Frame Sampler thread):
Released at 1000 ms with a release count of 3.
Starts executing immediately because it has the highest priority among the released tasks.
Completes execution at 1010 ms with an execution time of 10 ms.

(b) Task 2 (Time-stamp with Image Analysis thread):
Released at 1011 ms with a release count of 1.
Starts executing after Task 1 completes because it has the next highest priority. Completes execution at 1021 ms with an execution time of 10 ms.

(c) Task 4 (Time-stamp Image Save to File thread):
Released at 1022 ms with a release count of 1.
Starts executing after Task 2 completes because it has the next highest priority. Completes execution at 1032 ms with an execution time of 10 ms.

(d) Sequencer Thread:
Runs at 1033 ms, indicating the 31st cycle of the sequencer.

(e) Task 6 (Send Time-stamped Image to Remote thread):
Released at 1034 ms with a release count of 1.
Starts executing after the sequencer thread completes because it has the next highest priority among the remaining tasks.
Completes execution at 1044 ms with an execution time of 10 ms.

In this scenario, with preemption disabled, the tasks are executed based on their priorities and release times. The task with the highest priority among the released tasks starts executing first, and it runs to completion before the next highest priority task begins execution.

The execution sequence follows this pattern:

(a) Task 1 (highest priority) executes and completes.

(b) Task 2 (next highest priority) executes and completes.

(c) Task 4 (next highest priority) executes and completes. Sequencer Thread runs.

(d) Task 6 (remaining highest priority) executes and completes.

The tasks with lower priorities wait for the higher priority tasks to complete before starting their execution. This non-preemptive behavior ensures that each task runs to completion without being interrupted by other tasks, even if they have higher priorities.

In this scenario, all the threads are released simultaneously, but their execution order is determined by their assigned priorities. The task with the highest priority (Task 1) starts executing first and runs to completion. Then, the task with the next highest priority (Task 6) begins execution, followed by Task 2, Task 3, Task 5, and finally Task 7.

The execution sequence follows the priority order:

(a) Task 1 (highest priority)

(b) Task 6

(c) Task 2

(d) Task 3

(e) Task 5

(f) Task 7 (lowest priority)

Each task runs to completion before the next task in the priority order starts executing. This behavior is consistent with the non-preemptive scheduling approach, where a task runs uninterrupted until it completes, even if higher priority tasks are released during its execution.

Let's compare the worst-case execution time (WCET), jitter, predictability, determinism, and other important scheduling parameters between the FreeRTOS and POSIX implementations based on the provided output.

Worst-Case Execution Time (WCET):

(a) FreeRTOS:
- Task 1: 11 ms
- Task 2: 11 ms
- Task 3: 10 ms
- Task 4: 10 ms
- Task 5: 11 ms
- Task 6: 10 ms
- Task 7: 10 ms

(b) POSIX:
- Task 1: 92.489014 ms (set to 100ms)
- Task 2: 85.020996 ms (set to 100ms)
- Task 3: 90.000977 ms (set to 100ms)
- Task 4: 86.266846 ms (set to 100ms)
- Task 5: 88.769043 ms (set to 100ms)
- Task 6: 177.165039 ms (set to 100ms)
- Task 7: 168.037109 ms (set to 100ms)

The WCET values for the POSIX implementation are significantly higher compared to the FreeRTOS implementation. This difference can be attributed to the overhead of the POSIX threading model and the impact of the Linux scheduler on task execution times. **Jitter:**

(a) FreeRTOS: The jitter in the FreeRTOS implementation appears to be minimal, as the execution times are consistent across different instances of each task.

(b) POSIX: The POSIX implementation exhibits higher jitter, as evident from the variations in execution times for each task. For example, Task 6 has a WCET of 177.165039 ms, while its average execution time is 168.942459 ms, indicating significant jitter.

Predictability and Determinism:

(a) FreeRTOS: The FreeRTOS implementation demonstrates better predictability and determinism due to the consistent execution times and minimal jitter observed across task instances.

(b) POSIX: The POSIX implementation shows reduced predictability and determinism compared to FreeRTOS. The higher jitter and variable execution times make it more challenging to guarantee strict timing constraints.

Task Execution Units:

(a) FreeRTOS: The number of execution units for each task is consistent with the expected values based on the task periods and the total execution time.

(b) POSIX: The number of execution units for each task is also consistent with the expected values, indicating that the tasks are being released and executed according to their specified periods.

Scheduler Overhead:

(a) FreeRTOS: The FreeRTOS scheduler is designed for real-time embedded systems and has low overhead, contributing to the lower WCET values and minimal jitter observed.

(b) POSIX: The POSIX implementation, running on a Linux system, has higher scheduler overhead due to the complexities of the Linux scheduler and the need to manage multiple processes and threads. This overhead can impact the predictability and determinism of task execution.

The differences in WCET, jitter, predictability, and determinism between the FreeRTOS and POSIX implementations can be attributed to several factors:

Real-Time Operating System (RTOS) vs. General-Purpose Operating System (GPOS):

(a) FreeRTOS is a real-time operating system specifically designed for embedded systems, with a focus on determinism and real-time performance.

(b) POSIX, in this case running on Linux, is a general-purpose operating system that prioritizes overall system performance and fairness rather than strict real-time guarantees.

Scheduler Design:

(a) FreeRTOS has a simple and efficient scheduler tailored for real-time tasks, with minimal overhead and deterministic behavior.

(b) The Linux scheduler, used in the POSIX implementation, is more complex and designed to handle a wide range of workloads, which can introduce additional overhead and variability in task execution times.

System Overhead:

(a) FreeRTOS, being a lightweight RTOS, has lower system overhead compared to Linux, which runs multiple processes and services in the background.

(b) The higher system overhead in Linux can interfere with the execution of real-time tasks and contribute to increased jitter and reduced predictability.

2.1.1 C, D, T

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	333 ms	10 ms	333 ms	13.337891 ms
Service_2	1000 ms	10 ms	1000 ms	12.876953 ms
Service_3	2000 ms	10 ms	2000 ms	25.176025 ms
Service_4	1000 ms	10 ms	1000 ms	25.923828 ms
Service_5	2000 ms	10 ms	2000 ms	24.409912 ms
Service_6	1000 ms	10 ms	1000 ms	26.864014 ms
Service_7	10000 ms	10 ms	1000 ms	68.493164 ms

Table 3: C, T, D, for 10ms load, and sequencer frequency set to 30Hz in POSIX

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	333 ms	100 ms	333 ms	97 ms
Service_2	1000 ms	100 ms	1000 ms	97 ms
Service_3	2000 ms	100 ms	2000 ms	97 ms
Service_4	1000 ms	100 ms	1000 ms	97 ms
Service_5	2000 ms	100 ms	2000 ms	97 ms
Service_6	1000 ms	100 ms	1000 ms	96 ms
Service_7	10000 ms	10 ms	1000 ms	96 ms

Table 4: C, T, D, for 100ms load, and sequencer frequency set to 30Hz in freeRTOS

2.1.2 Cheddar

2.2 TIVA freeRTOS similar to seqgen2x(300Hz)

Output

```
***** Task 1 wcet 13 total_exectution_time 1072 execution unit 90 *****

***** Task 2 wcet 13 total_exectution_time 351 execution unit 29 *****

***** Task 3 wcet 13 total_exectution_time 176 execution unit 14 *****

***** Task 4 wcet 13 total_exectution_time 361 execution unit 29 *****

***** Task 5 wcet 12 total_exectution_time 168 execution unit 14 *****

***** Task 6 wcet 12 total_exectution_time 345 execution unit 29 *****

***** Task 7 wcet 12 total_exectution_time 24 execution unit 2 *****
```

Here we can see that the load is set to again 10ms and we can see the similar behaviour in the output that this is more deterministic than the LINUX posix api, we are getting around +2 to 3 ms of jitter which is better than the Linux

Note that we have set sequencer frequency to 300Hz and increased all the frequencies by multiplier of 10.

2.2.1 C, T, D

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	33.3 ms	10 ms	333 ms	13.337891 ms
Service_2	100 ms	10 ms	1000 ms	12.876953 ms
Service_3	200 ms	10 ms	2000 ms	25.176025 ms
Service_4	100 ms	10 ms	1000 ms	25.923828 ms
Service_5	200 ms	10 ms	2000 ms	24.409912 ms
Service_6	100 ms	10 ms	1000 ms	26.864014 ms
Service_7	1000 ms	10 ms	1000 ms	68.493164 ms

Table 5: C, T, D, for 10ms load, and sequencer frequency set to 30Hz

2.2.2 Cheddar

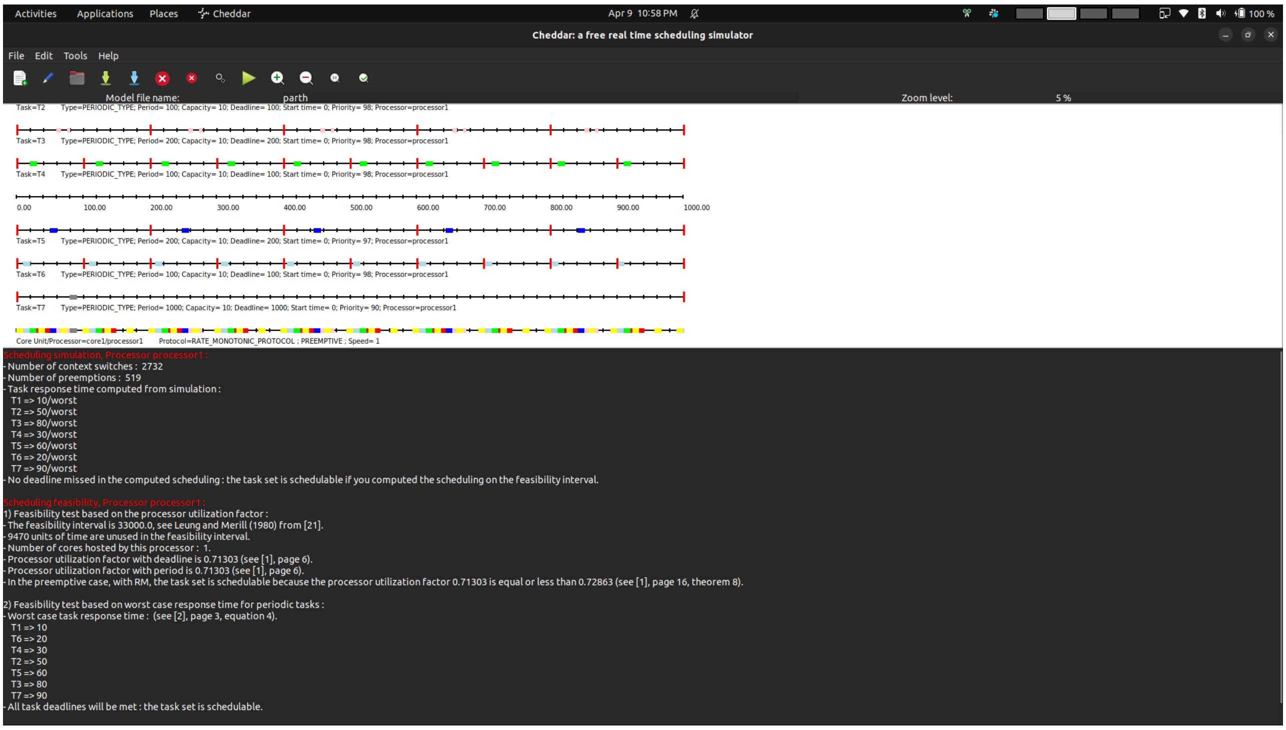

Figure 3: Cheddar Analysis for seqgen2 (300Hz)

2.3 Logs

logs can be found in the answer>logs>q2>seqgen.txt and answer>logs>q2>seqgen2.txt Code can be found in the answer>code>q2>seqgen.c and answer>code>q2>seqgen2.c

3 Question 3

Q: [40 points] Revise seqgen.c from both previous systems to increase the sequencer frequency and all service frequencies by a factor of 100 (3000 Hz). Build and execute the code under Linux and FreeRTOS on your target boards as before. For both operating systems determine the worst case execution time (WCET) for each service by printing or logging timestamps between two points in your code or by use of a profiling tool. Determine D, T, and C for each service and create an RM schedule in Cheddar using

your WCET estimates. Calculatethe % CPU utilization for these systems. Compare results between Linux and FreeRTOS in this higher-speed case.

Answer:

Code is same as the TivaWare freeRTOS code for seqgen just the logs are removed as it was taking some time and changed the multiple value to 1000.

In the provided output analysis, the multiplier has been increased to 100, meaning that all the services are running at 100 times their original frequency. Additionally, the execution time of each task has been reduced to 1 ms, and the UART printfs have been removed to minimize interference with task execution. Let's analyze the output in detail:
Increasing the frequency of the services can have several effects on the system:

- Higher CPU utilization: With tasks executing more frequently, the CPU utilization will increase as more time is spent executing the tasks.
- Increased context switching: As tasks are released and executed more often, there will be more context switches between tasks, which can introduce overhead and affect overall system performance.
- Potential resource contention: If tasks compete for shared resources (e.g., memory, I/O devices), the increased frequency of execution may lead to more resource contention and potential delays or blocking.
- Timing constraints: The system must ensure that the increased frequency of task execution does not violate any timing constraints or deadlines associated with the tasks.

System Behavior:

1. The increased frequency and reduced execution time of tasks result in a more fine-grained and responsive system.
2. The tasks are executed more frequently, allowing for faster processing and reaction to events.
3. However, the higher frequency also leads to increased context switching and potential resource contention, which may impact the overall system performance. and cause of that we needed to remove extra logs and keep the interference as low as possible.

The execution sequence follows the priority order:

1. Task 1 (highest priority)
2. Task 6
3. Task 2
4. Task 3
5. Task 5
6. Task 7 (lowest priority)

Each task runs to completion before the next task in the priority order starts executing. This behavior is consistent with the non-preemptive scheduling approach, where a task runs uninterrupted until it completes, even if higher priority tasks are released during its execution.

3.1 output

```
***** Task 1 wcet 2 total_exectution_time 9040 execution unit 9000 *****

***** Task 2 wcet 1 total_exectution_time 2999 execution unit 2999 *****

***** Task 3 wcet 1 total_exectution_time 1499 execution unit 1499 *****

***** Task 4 wcet 2 total_exectution_time 4802 execution unit 2999 *****
```

```
***** Task 5 wcet 1 total_exectution_time 1499 execution unit 1499 *****

***** Task 6 wcet 2 total_exectution_time 3272 execution unit 2999 *****

***** Task 7 wcet 1 total_exectution_time 299 execution unit 299 *****
```

Let's compare the worst-case execution time (WCET), jitter, predictability, determinism, and other important scheduling parameters between the FreeRTOS and POSIX implementations based on the provided output.

Worst-Case Execution Time (WCET):

1. FreeRTOS:
   - Task 1: 2 ms
   - Task 2: 1 ms
   - Task 3: 1 ms
   - Task 4: 2 ms
   - Task 5: 1 ms
   - Task 6: 2 ms
   - Task 7: 1 ms

3.1.1 C, T, D

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	3.33 ms	1 ms	33 ms	2 ms
Service_2	10 ms	1 ms	100 ms	1 ms
Service_3	20 ms	1 ms	200 ms	1 ms
Service_4	10 ms	1 ms	100 ms	2 ms
Service_5	20 ms	1 ms	200 ms	1 ms
Service_6	10 ms	1 ms	100 ms	2 ms
Service_7	100 ms	1 ms	100 ms	1 ms

Table 6: C, T, D, for 10ms load, and sequencer frequency set to 3000Hz

Service	Service Deadline (D)	Computation Time (C)	Period (T)	WCET
Service_1	3.3 ms	1 ms	3.3 ms	1.700195
Service_2	10 ms	1 ms	10 ms	1.650146
Service_3	20 ms	1 ms	20 ms	3.070801 ms
Service_4	10 ms	1 ms	10 ms	3.210938 ms
Service_5	20 ms	1 ms	20 ms	10.370117 ms
Service_6	10 ms	1 ms	10 ms	2.578125 ms
Service_7	100 ms	1 ms	10 ms	10.051025 ms

Table 7: C, T, D, for 1ms load in POSIX

Why is this happening ?? we have service one running at 3.33 ms and it has highest priority, service 2 and 6 has next highest priorities and we are releasing Task 2 first so Task 6 will run after task 2 and might get preempted by task 1. so WCET of T6 ¿ WCET T2 and Tasks 3,4,5 Has next highest priorities so It can be preempted by Task 1,2 and so their worst case execution time is more. And finally Service 7 It might and might note get preempted by Task 1 as the probability of getting preempted by task 7 is less (it runs every 100ms) but in our case it is getting preempted as we are running the code 180000 times. This explains the output.

3.1.2 Cheddar

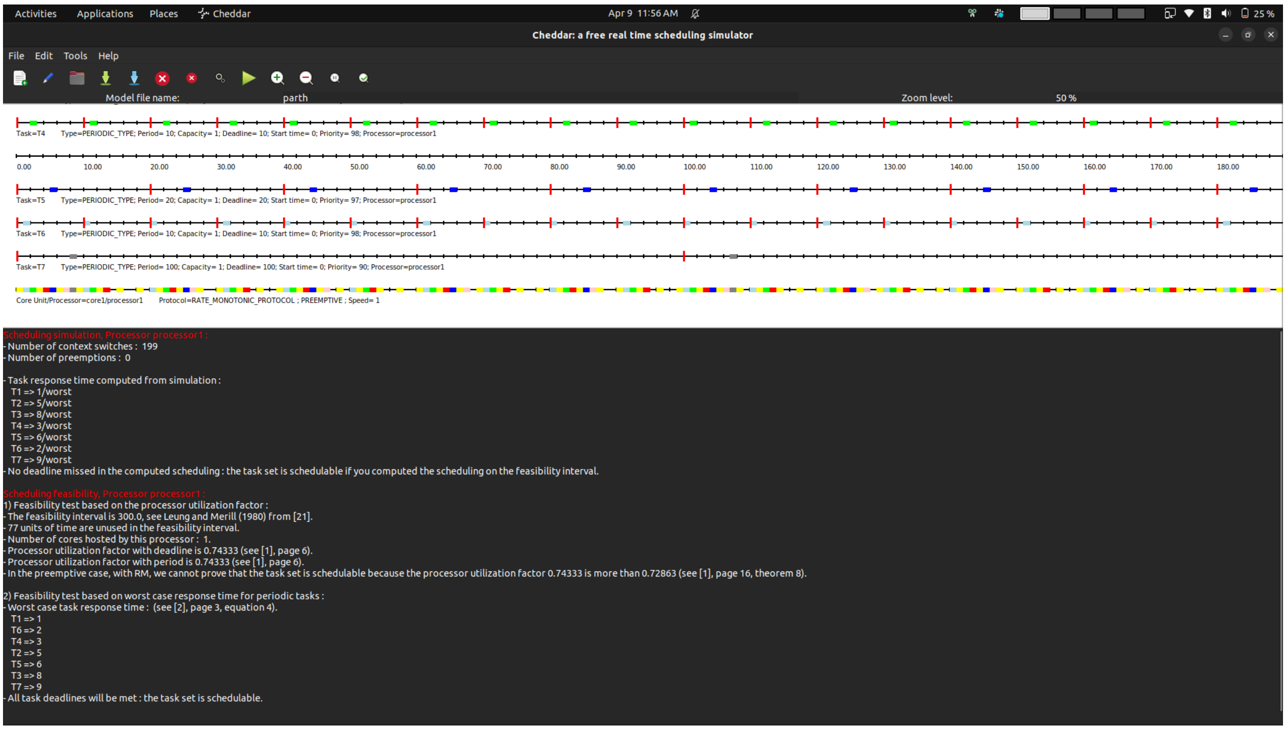

Figure 4: Cheddar Analysis for seqgen (30000Hz)

The processor utilization factor is calculated as 0.74333, which is more than the Liu & Layland bound of 0.72863 for the preemptive case.
Since the utilization factor exceeds the bound, the simulator cannot prove that the task set is schedulable using this test. but since all the deadlines are met we can say that the tasks are feasible and schedulable.

3.2 conclusion

For high frequency cases, we saw that the TIVA board gave more predictable results compared to the Jetson board. Even though the TIVA board's update rate was set to a slower 1ms, it performed better than the Jetson board. The Jetson board has multiple cores and a higher clock speed. FreeRTOS is lightweight and ideal for real-time applications. It works better than Linux, which is meant for desktop applications. Using a real-time patch for Linux could improve its predictability. However, Linux runs background processes that may affect the application's predictability. There are many factors to consider when designing a hard real-time system. In conclusion, for higher frequencies, FreeRTOS proves to be more predictable than Linux.

3.3 Logs

logs can be found in the answer>logs>q3>seqgen.txt
Code can be found in the answer>code>q3>seqgen.c

4 References

1. ECEN 5623 Lecture slides material and example codes.

2. REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS with LINUX and RTOS, Sam Siewert John Pratt (Chapter 6, 7 & 8).

3. Exercise 5 requirements included links and documentation.

Appendices

A C Code for the Implementation

main.c

```c

#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"
#include "limits.h"

#define FIB_LIMIT_FOR_32_BIT 47
#define ITERATION 120
#define MULTIPLIER 100
#define Hz (30 * MULTIPLIER)        // Hz
#define SEQUENCER_COUNT (900 * MULTIPLIER)
#define UART_BAUD_RATE 1000000

SemaphoreHandle_t task_1_SyncSemaphore, task_2_SyncSemaphore, task_3_SyncSemaphore,
task_4_SyncSemaphore, task_5_SyncSemaphore, task_6_SyncSemaphore,
task_7_SyncSemaphore;
TickType_t startTimeTick;
TaskHandle_t Task1_handle, Task2_handle, Task3_handle, Task4_handle, Task5_handle,
Task6_handle, Task7_handle;
volatile uint32_t counter_isr = 0;
uint32_t ulPeriod;
volatile bool abort_test = false;
uint32_t wcet[7];
uint32_t execution_time[7];
uint32_t execution_cycle[7];

void init_Timer();
void init_Uart();
void init_Clock();

void fibonacci()
{
    uint32_t i,j;
    uint32_t fib = 1, fib_a = 1, fib_b = 1;
    for ( i=0; i<ITERATION; i++)
```

```c
    {
        for(j=0; j<FIB_LIMIT_FOR_32_BIT; j++){
            fib_a = fib_b;
            fib_b = fib;
            fib = fib_a + fib_b;
        }

    }
}


void print_data(){
    uint32_t i = 0;
    for (i = 0; i < 7; i++){
        UARTprintf("***** Task %d wcet %d total_exectution_time %d execution unit %d *****\n\r", i+1, wcet[i], execution_time[i], execution_cycle[i]);
    }
}

void Timer0Isr_Sequencer(void)
{
    TickType_t xCurrentTick = xTaskGetTickCount();
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
    counter_isr++;

//    UARTprintf("Sequencer Thread ran at %d ms and Cycle of sequencer %d \n\r", xCurrentTick, counter_isr);

    if ((counter_isr % 10) == 0)
    {
        // Service_1 = RT_MAX-1 @ 300 Hz
        xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
    }

    if ((counter_isr % 30) == 0)
    {
        // Service_2 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis thread
        // Service_4 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
        // Service_6 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote thread

    }

    if ((counter_isr % 60) == 0)
    {
        // Service_3 = RT_MAX-3 @ 50 Hz
        xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
        // Service_5 = RT_MAX-3 @ 50 Hz
        xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
    }


    if ((counter_isr % 300) == 0)
```

```c
    {
        // Service_7 = RT_MIN   10 Hz
        xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
    }

    if (counter_isr > SEQUENCER_COUNT)
    {
        xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
        xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis thread
        xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
        xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
        xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
        xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote thread
        xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
        abort_test = true;
        ROM_TimerDisable(TIMER0_BASE, TIMER_A);
        print_data();
    }
}

// Process 1
void xTask1(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_1_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[0]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T1 S:%d, R %d\n\r", xCurrentTick, execution_cycle[0]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[0] += total_time;
            if(wcet[0] < total_time) wcet[0] = total_time;

            UARTprintf("T1 C:%d, E:%d\n\r", xFibTime, total_time);
        }
    }
    vTaskDelete( NULL );
}

void xTask2(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {
        xResult = xSemaphoreTake(task_2_SyncSemaphore, portMAX_DELAY);

```

```c

        if (xResult == pdPASS)
        {
            execution_cycle[1]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T2 S:%d, R %d\n\r", xCurrentTick, execution_cycle[1]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[1] += total_time;
            if(wcet[1] < total_time) wcet[1] = total_time;
            UARTprintf("T2 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskSuspend( NULL );
}
void xTask3(void *pvParameters)
{
    BaseType_t xResult;;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_3_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[2]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T3 S:%d, R %d\n\r", xCurrentTick, execution_cycle[2]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[2] += total_time;
            if(wcet[2] < total_time) wcet[2] = total_time;
            UARTprintf("T3 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask4(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_4_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[3]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T4 S:%d, R %d\n\r", xCurrentTick, execution_cycle[3]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
```

```c
                TickType_t total_time = (xFibTime - xCurrentTick);
                execution_time[3] += total_time;
                if(wcet[3] < total_time) wcet[3] = total_time;
                UARTprintf("T4 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
            }
        }
        vTaskDelete( NULL );
}
void xTask5(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_5_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[4]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T5 S:%d, R %d\n\r", xCurrentTick, execution_cycle[4]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[4] += total_time;
            if(wcet[4] < total_time) wcet[4] = total_time;
            UARTprintf("T5 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask6(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_6_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[5]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T6 S:%d, R %d\n\r", xCurrentTick, execution_cycle[5]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[5] += total_time;
            if(wcet[5] < total_time) wcet[5] = total_time;
            UARTprintf("T6 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
```

```c
void xTask7(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_7_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[6]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("T7 S:%d , R %d\n\r", xCurrentTick, execution_cycle[6]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[6] += total_time;
            if(wcet[6] < total_time) wcet[6] = total_time;
            UARTprintf("T7 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}

// Main function
int main(void)
{
    init_Clock();
    init_Uart();
    init_Timer();

    task_1_SyncSemaphore = xSemaphoreCreateBinary();
    task_2_SyncSemaphore = xSemaphoreCreateBinary();
    task_3_SyncSemaphore = xSemaphoreCreateBinary();
    task_4_SyncSemaphore = xSemaphoreCreateBinary();
    task_5_SyncSemaphore = xSemaphoreCreateBinary();
    task_6_SyncSemaphore = xSemaphoreCreateBinary();
    task_7_SyncSemaphore = xSemaphoreCreateBinary();

    UARTprintf("Cyclic executer : %d Hz\n\r", Hz);
    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 4, &Task1_handle);
    xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 3, &Task2_handle);
    xTaskCreate(xTask3, "Task3", configMINIMAL_STACK_SIZE, NULL, 2, &Task3_handle);
    xTaskCreate(xTask4, "Task4", configMINIMAL_STACK_SIZE, NULL, 3, &Task4_handle);
    xTaskCreate(xTask5, "Task5", configMINIMAL_STACK_SIZE, NULL, 2, &Task5_handle);
    xTaskCreate(xTask6, "Task6", configMINIMAL_STACK_SIZE, NULL, 3, &Task6_handle);
    xTaskCreate(xTask7, "Task7", configMINIMAL_STACK_SIZE, NULL, 1, &Task7_handle);

    startTimeTick = xTaskGetTickCount();

    vTaskStartScheduler();
    UARTprintf("\nTEST COMPLETE\n");
    return (0);
}
```

```
void init_Timer()
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);         // 32 bits Timer
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr_Sequencer); // Registering  isr

    ulPeriod = (SYSTEM_CLOCK / Hz);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 1);

    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
}

void init_Clock()
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
}

void init_Uart()
{
    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, UART_BAUD_RATE, SYSTEM_CLOCK);
}

/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```

seqgen.c

```
/* ========================================================================== */
/*                                                                            */
// Sam Siewert, December 2017
//
// Sequencer Generic
//
// The purpose of this code is to provide an example for how to best
// sequence a set of periodic services for problems similar to and including
// the final project in real-time systems.
//
// For example: Service_1 for camera frame aquisition
//              Service_2 for image analysis and timestamping
//              Service_3 for image processing (difference images)
//              Service_4 for save time-stamped image to file service
//              Service_5 for save processed image to file service
//              Service_6 for send image to remote server to save copy
//              Service_7 for elapsed time in syslog each minute for debug
//
// At least two of the services need to be real-time and need to run on a single
// core or run without affinity on the SMP cores available to the Linux
// scheduler as a group.  All services can be real-time, but you could choose
// to make just the first 2 real-time and the others best effort.
//
// For the standard project, to time-stamp images at the 1 Hz rate with unique
// clock images (unique second hand / seconds) per image, you might use the
// following rates for each service:
//
// Sequencer - 30 Hz
//                   [gives semaphores to all other services]
// Service_1 - 3 Hz  , every 10th Sequencer loop
//                   [buffers 3 images per second]
// Service_2 - 1 Hz  , every 30th Sequencer loop
//                   [time-stamp middle sample image with cvPutText or header]
// Service_3 - 0.5 Hz, every 60th Sequencer loop
//                   [difference current and previous time stamped images]
// Service_4 - 1 Hz, every 30th Sequencer loop
//                   [save time stamped image with cvSaveImage or write()]
// Service_5 - 0.5 Hz, every 60th Sequencer loop
//                   [save difference image with cvSaveImage or write()]
// Service_6 - 1 Hz, every 30th Sequencer loop
//                   [write current time-stamped image to TCP socket server]
// Service_7 - 0.1 Hz, every 300th Sequencer loop
//                   [syslog the time for debug]
//
// With the above, priorities by RM policy would be:
//
// Sequencer = RT_MAX   @ 30 Hz
// Servcie_1 = RT_MAX-1 @ 3 Hz
// Service_2 = RT_MAX-2 @ 1 Hz
// Service_3 = RT_MAX-3 @ 0.5 Hz
// Service_4 = RT_MAX-2 @ 1 Hz
// Service_5 = RT_MAX-3 @ 0.5 Hz
// Service_6 = RT_MAX-2 @ 1 Hz
```

```c
// Service_7 = RT_MIN   0.1 Hz
//
// Here are a few hardware/platform configuration settings on your Jetson
// that you should also check before running this code:
//
// 1) Check to ensure all your CPU cores on in an online state.
//
// 2) Check /sys/devices/system/cpu or do lscpu.
//
//    Tegra is normally configured to hot-plug CPU cores, so to make all
//    available, as root do:
//
//    echo 0 > /sys/devices/system/cpu/cpuquiet/tegra_cpuquiet/enable
//    echo 1 > /sys/devices/system/cpu/cpu1/online
//    echo 1 > /sys/devices/system/cpu/cpu2/online
//    echo 1 > /sys/devices/system/cpu/cpu3/online
//
// 3) Check for precision time resolution and support with cat /proc/timer_list
//
// 4) Ideally all printf calls should be eliminated as they can interfere with
//    timing.  They should be replaced with an in-memory event logger or at
//    least calls to syslog.
//
// 5) For simplicity, you can just allow Linux to dynamically load balance
//    threads to CPU cores (not set affinity) and as long as you have more
//    threads than you have cores, this is still an over-subscribed system
//    where RM policy is required over the set of cores.

// This is necessary for CPU affinity macros in Linux
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <pthread.h>
#include <sched.h>
#include <time.h>
#include <semaphore.h>

#include <syslog.h>
#include <sys/time.h>

#include <errno.h>

#define USEC_PER_MSEC (1000)
#define MS_PER_SEC (1000)
#define NANOSEC_PER_SEC (1000000000)
#define NUM_CPU_CORES (1)
#define TRUE (1)
#define FALSE (0)
#define ITERATION_COUNT 466500 // 100 ms load

#define NUM_THREADS (7 + 1)

int abortTest = FALSE;
```

```c
int abortS1 = FALSE, abortS2 = FALSE, abortS3 = FALSE, abortS4 = FALSE, abortS5 = FALSE, abortS6 = FALSE, abortS7 = FALSE;
sem_t semS1, semS2, semS3, semS4, semS5, semS6, semS7;
struct timeval start_time_val;

double wcet[7];
double execution_time[7];
int execution_cycle[7];

typedef struct
{
    int threadIdx;
    unsigned long long sequencePeriods;
} threadParams_t;

void *Sequencer(void *threadp);

void *Service_1(void *threadp);
void *Service_2(void *threadp);
void *Service_3(void *threadp);
void *Service_4(void *threadp);
void *Service_5(void *threadp);
void *Service_6(void *threadp);
void *Service_7(void *threadp);
double getTimeMsec(void);
void print_scheduler(void);

#define FIB_LIMIT_FOR_32_BIT 47
#define ITERATION_COUNT_FIB 15000

void fibTest(int interation_count)
{
    int fib, fib0, fib1;
    int jdx = 0;
    for (int idx = 0; idx < interation_count; idx++)
    {
        fib = fib0 + fib1;
        while (jdx < FIB_LIMIT_FOR_32_BIT)
        {
            fib0 = fib1;
            fib1 = fib;
            fib = fib0 + fib1;
            jdx++;
        }
        jdx = 0;
    }
}


void print_data(){
    for(int i=0; i<7; i++){
        syslog(LOG_CRIT, "**** Task %d):  WCET: %f, total_execution time : %f, execution cycles : %d, average execution time : %f **** \n ", i+1, wcet[i], execution_time[i], execution_cycle[i], execution_time[i]/execution_cycle[i]);
        printf("**** Task %d):  WCET: %f, total_execution time : %f, execution cycles : %d, average execution time : %f **** \n ", i+1, wcet[i], execution_time[i], execution_cycle[i], execution_time[i]/execution_cycle[i]);
```

```
    }

}

double read_time(double *var)
{
    struct timeval tv;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("readTOD");
        return 0.0;
    }
    else
    {
        *var = ((double)(((double)tv.tv_sec * 1000) + (((double)tv.tv_usec) / 1000.0)));
    }
    return (*var);
}


void main(void)
{
    struct timeval current_time_val;
    int i, rc, scope;
    cpu_set_t threadcpu;
    pthread_t threads[NUM_THREADS];
    threadParams_t threadParams[NUM_THREADS];
    pthread_attr_t rt_sched_attr[NUM_THREADS];
    int rt_max_prio, rt_min_prio;
    struct sched_param rt_param[NUM_THREADS];
    struct sched_param main_param;
    pthread_attr_t main_attr;
    pid_t mainpid;
    cpu_set_t allcpuset;

    printf("Starting Sequencer Demo\n");
    syslog(LOG_CRIT, "Starting Sequencer Demo\n");

    printf("testing Fib load with iterations :%d\n", ITERATION_COUNT);
    double avg_time = 0;
    for(int i=0;i<10;i++){
        double start, end;
        read_time(&start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        double total_ex = end - start;
        avg_time += total_ex;
        printf("iteration %d) Start time: %f ms , end time: %f ms , execution time: %f ms\n\n",i, start, end, total_ex);
        syslog(LOG_CRIT, "iteration %d) Start time: %f ms , end time: %f ms , execution time: %f ms\n\n",i, start, end, total_ex);

    }
    printf("***** Average time %f *****\n", avg_time / 10);
    syslog(LOG_CRIT, "***** Average time %f *****\n", avg_time / 10);

```

```c

    gettimeofday(&start_time_val, (struct timezone *)0);
    gettimeofday(&current_time_val, (struct timezone *)0);
    syslog(LOG_CRIT, "Sequencer @ sec=%d, msec=%d\n", (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);
    printf("Sequencer @ sec=%d, msec=%d\n", (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);

    printf("System has %d processors configured and %d available.\n", get_nprocs_conf(), get_nprocs());
    syslog(LOG_CRIT, "System has %d processors configured and %d available.\n", get_nprocs_conf(), get_nprocs());

    CPU_ZERO(&allcpuset);

    for (i = 0; i < NUM_CPU_CORES; i++)
        CPU_SET(i, &allcpuset);

    printf("Using CPUS=%d from total available.\n", CPU_COUNT(&allcpuset));

    // initialize the sequencer semaphores
    //
    if (sem_init(&semS1, 0, 0))
    {
        printf("Failed to initialize S1 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS2, 0, 0))
    {
        printf("Failed to initialize S2 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS3, 0, 0))
    {
        printf("Failed to initialize S3 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS4, 0, 0))
    {
        printf("Failed to initialize S4 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS5, 0, 0))
    {
        printf("Failed to initialize S5 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS6, 0, 0))
    {
        printf("Failed to initialize S6 semaphore\n");
        exit(-1);
    }
    if (sem_init(&semS7, 0, 0))
    {
        printf("Failed to initialize S7 semaphore\n");
        exit(-1);
    }
```

```c

    mainpid = getpid();

    rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    rt_min_prio = sched_get_priority_min(SCHED_FIFO);

    rc = sched_getparam(mainpid, &main_param);
    main_param.sched_priority = rt_max_prio;
    rc = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
    if (rc < 0)
        perror("main_param");
    print_scheduler();

    pthread_attr_getscope(&main_attr, &scope);

    if (scope == PTHREAD_SCOPE_SYSTEM)
        printf("PTHREAD SCOPE SYSTEM\n");
    else if (scope == PTHREAD_SCOPE_PROCESS)
        printf("PTHREAD SCOPE PROCESS\n");
    else
        printf("PTHREAD SCOPE UNKNOWN\n");

    printf("rt_max_prio=%d\n", rt_max_prio);
    printf("rt_min_prio=%d\n", rt_min_prio);

    for (i = 0; i < NUM_THREADS; i++)
    {

        CPU_ZERO(&threadcpu);
        CPU_SET(3, &threadcpu);

        rc = pthread_attr_init(&rt_sched_attr[i]);
        rc = pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
        rc = pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
        rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &threadcpu);

        rt_param[i].sched_priority = rt_max_prio - i;
        pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);

        threadParams[i].threadIdx = i;
    }

    printf("Service threads will run on %d CPU cores\n", CPU_COUNT(&threadcpu));
    syslog(LOG_CRIT, "Service threads will run on %d CPU cores\n", CPU_COUNT(&threadcpu));

    // Create Service threads which will block awaiting release for:
    //

    // Servcie_1 = RT_MAX-1 @ 3 Hz
    //
    rt_param[1].sched_priority = rt_max_prio - 1;
    pthread_attr_setschedparam(&rt_sched_attr[1], &rt_param[1]);
    rc = pthread_create(&threads[1],       // pointer to thread descriptor
                        &rt_sched_attr[1], // use specific attributes
                        //(void *)0,               // default attributes
```

```
                             Service_1,                    // thread function entry point
                             (void *)&(threadParams[1]) // parameters to pass in
    );
    if (rc < 0)
        perror("pthread_create for service 1");
    else{
        printf("pthread_create successful for service 1\n");
        syslog(LOG_CRIT, "pthread_create successful for service 1\n");
    }

    // Service_2 = RT_MAX-2 @ 1 Hz
    //
    rt_param[2].sched_priority = rt_max_prio - 2;
    pthread_attr_setschedparam(&rt_sched_attr[2], &rt_param[2]);
    rc = pthread_create(&threads[2], &rt_sched_attr[2], Service_2, (void *)&
(threadParams[2]));
    if (rc < 0)
        perror("pthread_create for service 2");
    else{
        printf("pthread_create successful for service 2\n");
        syslog(LOG_CRIT, "pthread_create successful for service 2\n");
    }

    // Service_3 = RT_MAX-3 @ 0.5 Hz
    //
    rt_param[3].sched_priority = rt_max_prio - 3;
    pthread_attr_setschedparam(&rt_sched_attr[3], &rt_param[3]);
    rc = pthread_create(&threads[3], &rt_sched_attr[3], Service_3, (void *)&
(threadParams[3]));
    if (rc < 0)
        perror("pthread_create for service 3");
    else{
        printf("pthread_create successful for service 3\n");
        syslog(LOG_CRIT, "pthread_create successful for service 3\n");
    }

    // Service_4 = RT_MAX-2 @ 1 Hz
    //
    rt_param[4].sched_priority = rt_max_prio - 2;
    pthread_attr_setschedparam(&rt_sched_attr[4], &rt_param[4]);
    rc = pthread_create(&threads[4], &rt_sched_attr[4], Service_4, (void *)&
(threadParams[4]));
    if (rc < 0)
        perror("pthread_create for service 4");
    else{
        printf("pthread_create successful for service 4\n");
        syslog(LOG_CRIT, "pthread_create successful for service 4\n");
    }

    // Service_5 = RT_MAX-3 @ 0.5 Hz
    //
    rt_param[5].sched_priority = rt_max_prio - 3;
    pthread_attr_setschedparam(&rt_sched_attr[5], &rt_param[5]);
    rc = pthread_create(&threads[5], &rt_sched_attr[5], Service_5, (void *)&
(threadParams[5]));
    if (rc < 0)
        perror("pthread_create for service 5");
```

```
    else{

        printf("pthread_create successful for service 5\n");
        syslog(LOG_CRIT, "pthread_create successful for service 5\n");
    }

    // Service_6 = RT_MAX-2 @ 1 Hz
    //
    rt_param[6].sched_priority = rt_max_prio - 2;
    pthread_attr_setschedparam(&rt_sched_attr[6], &rt_param[6]);
    rc = pthread_create(&threads[6], &rt_sched_attr[6], Service_6, (void *)&(threadParams[6]));
    if (rc < 0)
        perror("pthread_create for service 6");
    else{

        syslog(LOG_CRIT, "pthread_create successful for service 6\n");
    }

    // Service_7 = RT_MIN   0.1 Hz
    //
    rt_param[7].sched_priority = rt_min_prio;
    pthread_attr_setschedparam(&rt_sched_attr[7], &rt_param[7]);
    rc = pthread_create(&threads[7], &rt_sched_attr[7], Service_7, (void *)&(threadParams[7]));
    if (rc < 0)
        perror("pthread_create for service 7");
    else{

        printf("pthread_create successful for service 7\n");
        syslog(LOG_CRIT, "pthread_create successful for service 7\n");
    }

    // Wait for service threads to initialize and await release by sequencer.
    //
    // Note that the sleep is not necessary of RT service threads are created wtih
    // correct POSIX SCHED_FIFO priorities compared to non-RT priority of this main
    // program.
    //
    // usleep(1000000);

    // Create Sequencer thread, which like a cyclic executive, is highest prio
    printf("Start sequencer\n");
    syslog(LOG_CRIT, "Start sequencer\n");
    threadParams[0].sequencePeriods = 900;

    // Sequencer = RT_MAX   @ 30 Hz
    //
    rt_param[0].sched_priority = rt_max_prio;
    pthread_attr_setschedparam(&rt_sched_attr[0], &rt_param[0]);
    rc = pthread_create(&threads[0], &rt_sched_attr[0], Sequencer, (void *)&(threadParams[0]));
    if (rc < 0)
        perror("pthread_create for sequencer service 0");
    else{

        printf("pthread_create successful for sequeencer service 0\n");
```

```c
        syslog(LOG_CRIT, "pthread_create successful for sequeencer service 0\n");
    }

    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);

    printf("\nTEST COMPLETE\n");
    syslog(LOG_CRIT, "\nTEST COMPLETE\n");
}

void *Sequencer(void *threadp)
{
    struct timeval current_time_val;
    struct timespec delay_time = {0, 33333333}; // delay for 33.33 msec, 30 Hz
    struct timespec remaining_time;
    double current_time;
    double residual;
    int rc, delay_cnt = 0;
    unsigned long long seqCnt = 0;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    gettimeofday(&current_time_val, (struct timezone *)0);
    syslog(LOG_CRIT, "Sequencer thread @ sec=%d, msec=%d\n", (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);
    printf("Sequencer thread @ sec=%d, msec=%d\n", (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);

    do
    {
        delay_cnt = 0;
        residual = 0.0;

        gettimeofday(&current_time_val, (struct timezone *)0);
        syslog(LOG_CRIT, "Sequencer thread prior to delay @ sec=%d, msec=%d\n", (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);

        do
        {
            rc = nanosleep(&delay_time, &remaining_time);

            if (rc == EINTR)
            {
                residual = remaining_time.tv_sec + ((double)remaining_time.tv_nsec / (double)NANOSEC_PER_SEC);

                if (residual > 0.0)
                    printf("residual=%lf, sec=%d, nsec=%d\n", residual, (int)remaining_time.tv_sec, (int)remaining_time.tv_nsec);

                delay_cnt++;
            }
            else if (rc < 0)
            {
                perror("Sequencer nanosleep");
                exit(-1);
            }
```

```c

        } while ((residual > 0.0) && (delay_cnt < 100));

        seqCnt++;
        gettimeofday(&current_time_val, (struct timezone *)0);
        syslog(LOG_CRIT, "Sequencer cycle %llu @ sec=%d, msec=%d\n", seqCnt, (int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec / USEC_PER_MSEC);

        if (delay_cnt > 1)
            printf("Sequencer looping delay %d\n", delay_cnt);

        // Release each service at a sub-rate of the generic sequencer rate

        // Servcie_1 = RT_MAX-1 @ 3 Hz
        if ((seqCnt % 10) == 0)
        {
            syslog(LOG_CRIT, "Task 1 (Frame Sampler thread) Released \n");
            sem_post(&semS1); // Frame Sampler thread
        }

        // Service_2 = RT_MAX-2 @ 1 Hz
        if ((seqCnt % 30) == 0)
        {
            syslog(LOG_CRIT, "Task 2 (Time-stamp with Image Analysis thread) Released \n");
            sem_post(&semS2); // Time-stamp with Image Analysis thread
        }

        // Service_3 = RT_MAX-3 @ 0.5 Hz
        if ((seqCnt % 60) == 0)
        {
            syslog(LOG_CRIT, "Task 3 ( Difference Image Proc thread) Released \n");
            sem_post(&semS3); // Difference Image Proc thread
        }

        // Service_4 = RT_MAX-2 @ 1 Hz
        if ((seqCnt % 30) == 0)
        {
            syslog(LOG_CRIT, "Task 4 (Time-stamp Image Save to File thread) Released \n");
            sem_post(&semS4); // Time-stamp Image Save to File thread
        }

        // Service_5 = RT_MAX-3 @ 0.5 Hz
        if ((seqCnt % 60) == 0)
        {
            syslog(LOG_CRIT, "Task 5 (Processed Image Save to File thread) Released \n");
            sem_post(&semS5); // Processed Image Save to File thread
        }

        // Service_6 = RT_MAX-2 @ 1 Hz
        if ((seqCnt % 30) == 0)
        {
            syslog(LOG_CRIT, "Task 6 (Send Time-stamped Image to Remote thread) Released \n");
```

```
            sem_post(&semS6); // Send Time-stamped Image to Remote thread
        }

        // Service_7 = RT_MIN   0.1 Hz
        if ((seqCnt % 300) == 0)
        {
            syslog(LOG_CRIT, "Task 7 (10 sec Tick Debug thread) Released \n");
            sem_post(&semS7); // 10 sec Tick Debug thread
        }

        gettimeofday(&current_time_val, NULL);
        syslog(LOG_CRIT, "Sequencer release all sub-services @ sec=%d, msec=%d\n",
(int)(current_time_val.tv_sec - start_time_val.tv_sec), (int)current_time_val.tv_usec
/ USEC_PER_MSEC);

    } while (!abortTest && (seqCnt < threadParams->sequencePeriods));

    sem_post(&semS1);
    sem_post(&semS2);
    sem_post(&semS3);
    sem_post(&semS4);
    sem_post(&semS5);
    sem_post(&semS6);
    sem_post(&semS7);
    abortS1 = TRUE;
    abortS2 = TRUE;
    abortS3 = TRUE;
    abortS4 = TRUE;
    abortS5 = TRUE;
    abortS6 = TRUE;
    abortS7 = TRUE;
    print_data();

    pthread_exit((void *)0);
}

void *Service_1(void *threadp)
{
    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 1, Frame Sampler thread @ msec=%f \n", start);
    printf("Task 1, Frame Sampler thread @ msec=%f \n", start);

    while (!abortS1)
    {
        sem_wait(&semS1);

        execution_cycle[0]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 1, Frame Sampler start %d @ msec=%f",
execution_cycle[0], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[0]) wcet[0] = total;
```

```
        execution_time[0] += total;
        syslog(LOG_CRIT, "Task 1, Frame Sampler Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);
}

void *Service_2(void *threadp)
{

    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread @ msec=%f \n", start);
    printf("Task 2, Time-stamp with Image Analysis thread @ msec=%f \n", start);

    while (!abortS2)
    {
        sem_wait(&semS2);

        execution_cycle[1]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread start %d @ msec=%f", execution_cycle[1], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[1]) wcet[1] = total;
        execution_time[1] += total;
        syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);

}

void *Service_3(void *threadp)
{

    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 3, Difference Image Proc thread @ msec=%f \n", start);
    printf("Task 3, Difference Image Proc thread @ msec=%f \n", start);

    while (!abortS3)
    {
        sem_wait(&semS3);

        execution_cycle[2]++;
        read_time(&start);
```

```c
        syslog(LOG_CRIT, "Task 3, Difference Image Proc  start %d @ msec=%f", execution_cycle[2], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[2]) wcet[2] = total;
        execution_time[2] += total;
        syslog(LOG_CRIT, "Task 3, Difference Image Proc Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);
}

void *Service_4(void *threadp)
{

    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File thread @ msec=%f \n", start);
    printf("Task 4, Time-stamp Image Save to File thread @ msec=%f \n", start);

    while (!abortS4)
    {
        sem_wait(&semS4);

        execution_cycle[3]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File start %d @ msec=%f", execution_cycle[3], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[3]) wcet[3] = total;
        execution_time[3] += total;
        syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);

}

void *Service_5(void *threadp)
{

    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 5, Processed Image Save to File thread @ msec=%f \n", start);
    printf("Task 5, Processed Image Save to File thread @ msec=%f \n", start);

```

```
    while (!abortS5)
    {
        sem_wait(&semS5);

        execution_cycle[4]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 5, Processed Image Save to File start %d @ msec=%f", execution_cycle[4], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[4]) wcet[4] = total;
        execution_time[4] += total;
        syslog(LOG_CRIT, "Task 5, Processed Image Save to File Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);


}

void *Service_6(void *threadp)
{

    double start, end, total;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote thread @ msec=%f \n", start);
    printf("Task 6, Send Time-stamped Image to Remote thread @ msec=%f \n", start);

    while (!abortS6)
    {
        sem_wait(&semS6);

        execution_cycle[5]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote start %d @ msec=%f", execution_cycle[5], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[5]) wcet[5] = total;
        execution_time[5] += total;
        syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote Execution complete @ msec=%f, execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);

}

void *Service_7(void *threadp)
{
    double start, end, total;
```

```
    threadParams_t *threadParams = (threadParams_t *)threadp;

    read_time(&start);
    syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug thread @ msec=%f \n", start);
    printf("Task 7, 10 sec Tick Debug Thread @ msec=%f \n", start);

    while (!abortS7)
    {
        sem_wait(&semS7);

        execution_cycle[6]++;
        read_time(&start);
        syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug start %d @ msec=%f",
execution_cycle[6], start);
        fibTest(ITERATION_COUNT);
        read_time(&end);
        total = end - start;
        if(total > wcet[6]) wcet[6] = total;
        execution_time[6] += total;
        syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug Execution complete @ msec=%f,
execution time : %f ms\n", end, total);
    }

    pthread_exit((void *)0);
}

double getTimeMsec(void)
{
    struct timespec event_ts = {0, 0};

    clock_gettime(CLOCK_MONOTONIC, &event_ts);
    return ((event_ts.tv_sec) * 1000.0) + ((event_ts.tv_nsec) / 1000000.0);
}

void print_scheduler(void)
{
    int schedType;

    schedType = sched_getscheduler(getpid());

    switch (schedType)
    {
    case SCHED_FIFO:
        printf("Pthread Policy is SCHED_FIFO\n");
        break;
    case SCHED_OTHER:
        printf("Pthread Policy is SCHED_OTHER\n");
        exit(-1);
        break;
    case SCHED_RR:
        printf("Pthread Policy is SCHED_RR\n");
        exit(-1);
        break;
    default:
        printf("Pthread Policy is UNKNOWN\n");
        exit(-1);
    }
```

```
}

```

main.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"
#include "limits.h"

#define FIB_LIMIT_FOR_32_BIT 47
#define ITERATION 1200
#define MULTIPLIER 10
#define Hz (30 * MULTIPLIER)        // Hz
#define SEQUENCER_COUNT 900
#define UART_BAUD_RATE 1000000

SemaphoreHandle_t task_1_SyncSemaphore, task_2_SyncSemaphore, task_3_SyncSemaphore,
task_4_SyncSemaphore, task_5_SyncSemaphore, task_6_SyncSemaphore,
task_7_SyncSemaphore;
TickType_t startTimeTick;
TaskHandle_t Task1_handle, Task2_handle, Task3_handle, Task4_handle, Task5_handle,
Task6_handle, Task7_handle;
uint32_t counter_isr = 0;
uint32_t ulPeriod;
volatile bool abort_test = false;
uint32_t wcet[7];
uint32_t execution_time[7];
uint32_t execution_cycle[7];

void init_Timer();
void init_Uart();
void init_Clock();

void fibonacci()
{
    uint32_t i,j;
    uint32_t fib = 1, fib_a = 1, fib_b = 1;
    for ( i=0; i<ITERATION; i++)
```

```c
    {
        for(j=0; j<FIB_LIMIT_FOR_32_BIT; j++){
            fib_a = fib_b;
            fib_b = fib;
            fib = fib_a + fib_b;
        }

    }
}


void print_data(){
    uint32_t i = 0;
    for (i = 0; i < 7; i++){
        UARTprintf("***** Task %d wcet %d total_exectution_time %d execution unit %d *****\n\r", i+1, wcet[i], execution_time[i], execution_cycle[i]);
    }
}

void Timer0Isr_Sequencer(void)
{
    TickType_t xCurrentTick = xTaskGetTickCount();
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
    counter_isr++;

    UARTprintf("Sequencer Thread ran at %d ms and Cycle of sequencer %d \n\r", xCurrentTick, counter_isr);

    if ((counter_isr % 10) == 0)
    {
        // Service_1 = RT_MAX-1 @ 300 Hz
        xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
    }

    if ((counter_isr % 30) == 0)
    {
        // Service_2 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis thread
        // Service_4 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
        // Service_6 = RT_MAX-2 @ 100 Hz
        xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote thread

    }

    if ((counter_isr % 60) == 0)
    {
        // Service_3 = RT_MAX-3 @ 50 Hz
        xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
        // Service_5 = RT_MAX-3 @ 50 Hz
        xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
    }


    if ((counter_isr % 300) == 0)
```

```
    {
        // Service_7 = RT_MIN   10 Hz
        xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
    }

    if (counter_isr > SEQUENCER_COUNT)
    {
        xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
        xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis thread
        xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
        xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
        xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
        xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote thread
        xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
        abort_test = true;
        ROM_TimerDisable(TIMER0_BASE, TIMER_A);
        print_data();
    }
}

// Process 1
void xTask1(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_1_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[0]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 1 (Frame Sampler thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[0]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[0] += total_time;
            if(wcet[0] < total_time) wcet[0] = total_time;

            UARTprintf("Task 1 (Frame Sampler thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, total_time);
        }
    }
    vTaskDelete( NULL );
}

void xTask2(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {
```

```
        xResult = xSemaphoreTake(task_2_SyncSemaphore, portMAX_DELAY);


        if (xResult == pdPASS)
        {
            execution_cycle[1]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 2 (Time-stamp with Image Analysis thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[1]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[1] += total_time;
            if(wcet[1] < total_time) wcet[1] = total_time;
            UARTprintf("Task 2 (Time-stamp with Image Analysis thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskSuspend( NULL );
}
void xTask3(void *pvParameters)
{
    BaseType_t xResult;;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_3_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[2]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 3 (Difference Image Proc thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[2]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[2] += total_time;
            if(wcet[2] < total_time) wcet[2] = total_time;
            UARTprintf("Task 3 (Difference Image Proc thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask4(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_4_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
```

```c
            execution_cycle[3]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 4 (Time-stamp Image Save to File thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[3]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[3] += total_time;
            if(wcet[3] < total_time) wcet[3] = total_time;
            UARTprintf("Task 4 (Time-stamp Image Save to File thread) Completion Time:%d ms, Execution Timet:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask5(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_5_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[4]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 5 (Processed Image Save to File thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[4]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[4] += total_time;
            if(wcet[4] < total_time) wcet[4] = total_time;
            UARTprintf("Task 5 (Processed Image Save to File thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask6(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_6_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[5]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 6 (Send Time-stamped Image to Remote thread) Start Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[5]);
            fibonacci();
```

```c
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[5] += total_time;
            if(wcet[5] < total_time) wcet[5] = total_time;
            UARTprintf("Task 6 (Send Time-stamped Image to Remote thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}
void xTask7(void *pvParameters)
{
    BaseType_t xResult;

    while (!abort_test)
    {

        xResult = xSemaphoreTake(task_7_SyncSemaphore, portMAX_DELAY);

        if (xResult == pdPASS)
        {
            execution_cycle[6]++;
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 7 (10 sec Tick Debug thread) Start Time:%d ms , Release count %d \n\r", xCurrentTick, execution_cycle[6]);
            fibonacci();
            TickType_t xFibTime = xTaskGetTickCount();
            TickType_t total_time = (xFibTime - xCurrentTick);
            execution_time[6] += total_time;
            if(wcet[6] < total_time) wcet[6] = total_time;
            UARTprintf("Task 7 (10 sec Tick Debug thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
        }
    }
    vTaskDelete( NULL );
}

// Main function
int main(void)
{
    init_Clock();
    init_Uart();
    init_Timer();

    task_1_SyncSemaphore = xSemaphoreCreateBinary();
    task_2_SyncSemaphore = xSemaphoreCreateBinary();
    task_3_SyncSemaphore = xSemaphoreCreateBinary();
    task_4_SyncSemaphore = xSemaphoreCreateBinary();
    task_5_SyncSemaphore = xSemaphoreCreateBinary();
    task_6_SyncSemaphore = xSemaphoreCreateBinary();
    task_7_SyncSemaphore = xSemaphoreCreateBinary();

    UARTprintf("Cyclic executer : %d Hz\n\r", Hz);
    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 4, &Task1_handle);
    xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 3, &Task2_handle);
    xTaskCreate(xTask3, "Task3", configMINIMAL_STACK_SIZE, NULL, 2, &Task3_handle);
    xTaskCreate(xTask4, "Task4", configMINIMAL_STACK_SIZE, NULL, 3, &Task4_handle);
```

```
    xTaskCreate(xTask5, "Task5", configMINIMAL_STACK_SIZE, NULL, 2, &Task5_handle);
    xTaskCreate(xTask6, "Task6", configMINIMAL_STACK_SIZE, NULL, 3, &Task6_handle);
    xTaskCreate(xTask7, "Task7", configMINIMAL_STACK_SIZE, NULL, 1, &Task7_handle);

    startTimeTick = xTaskGetTickCount();

    vTaskStartScheduler();
    UARTprintf("\nTEST COMPLETE\n");
    return (0);
}

void init_Timer()
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);         // 32 bits Timer
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr_Sequencer); // Registering  isr

    ulPeriod = (SYSTEM_CLOCK / Hz);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 1);

    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
}

void init_Clock()
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
}

void init_Uart()
{
    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, UART_BAUD_RATE, SYSTEM_CLOCK);
}

/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```