

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 9, 2024

Contents

| | |
|--|-----------|
| List of Figures | 1 |
| List of Tables | 1 |
| 1 Question 1 | 3 |
| 1.1 API Explanation: | 3 |
| 1.2 Code Flow: | 3 |
| 1.3 Output Analysis: | 4 |
| 1.4 SEQGEN2x | 9 |
| 1.5 Comparison with Linux and FreeRTOS: | 10 |
| 1.6 D, C, and T Table for All Services: | 10 |
| 1.7 Cheddar Schedule and CPU Utilization: | 11 |
| 1.8 Logs | 12 |
| 2 Question 2 | 12 |
| 2.1 30 Hz sequencer | 12 |
| 2.1.1 C, D, T | 18 |
| 2.1.2 Cheddar | 18 |
| 2.2 TIVA freeRTOS similar to seqgen2x(300Hz) | 18 |
| 2.2.1 C, T, D | 18 |
| 2.2.2 Cheddar | 19 |
| 2.3 Logs | 19 |
| 3 Question 3 | 19 |
| 3.1 output | 20 |
| 3.1.1 C, T, D | 21 |
| 3.1.2 Cheddar | 22 |
| 3.2 conclusion | 22 |
| 3.3 Logs | 22 |
| 4 References | 22 |
| Appendices | 24 |
| A C Code for the Implementation | 24 |

List of Figures

| | | |
|---|---------------------------------------|----|
| 1 | Cheddar Analysis for seqgen (30Hz) | 11 |
| 2 | Cheddar Analysis for seqgen2 (300Hz) | 12 |
| 3 | Cheddar Analysis for seqgen2 (300Hz) | 19 |
| 4 | Cheddar Analysis for seqgen (30000Hz) | 22 |

List of Tables

| | | |
|---|---|----|
| 1 | C, T, D, for 30Hz and 100ms load | 11 |
| 2 | C, T, D, for 10ms load | 11 |
| 3 | C, T, D, for 10ms load, and sequencer frequency set to 30Hz | 18 |
| 4 | C, T, D, for 10ms load, and sequencer frequency set to 30Hz | 18 |
| 5 | C, T, D, for 10ms load, and sequencer frequency set to 3000Hz | 21 |

**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 ($1000 \text{ ms} / 33.33 \text{ ms} = 30 \text{ Hz}$).

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:

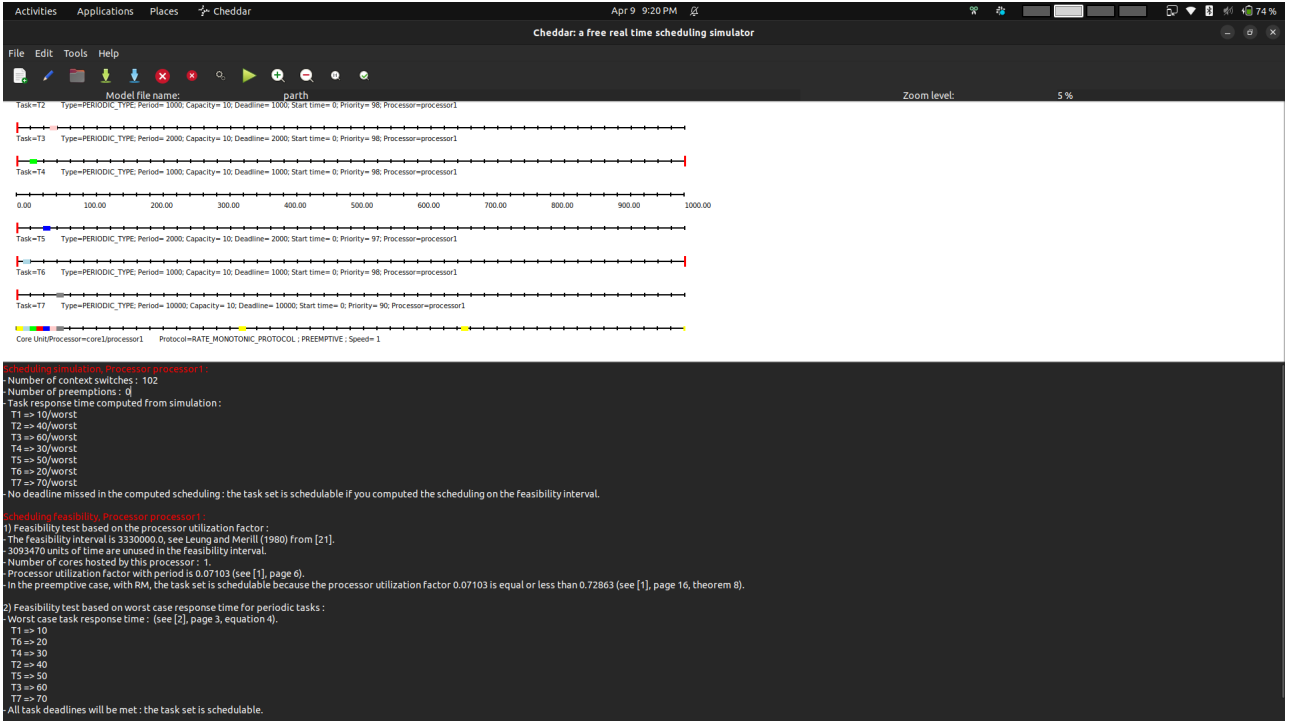


Figure 1: Cheddar Analysis for seqgen (30Hz)

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

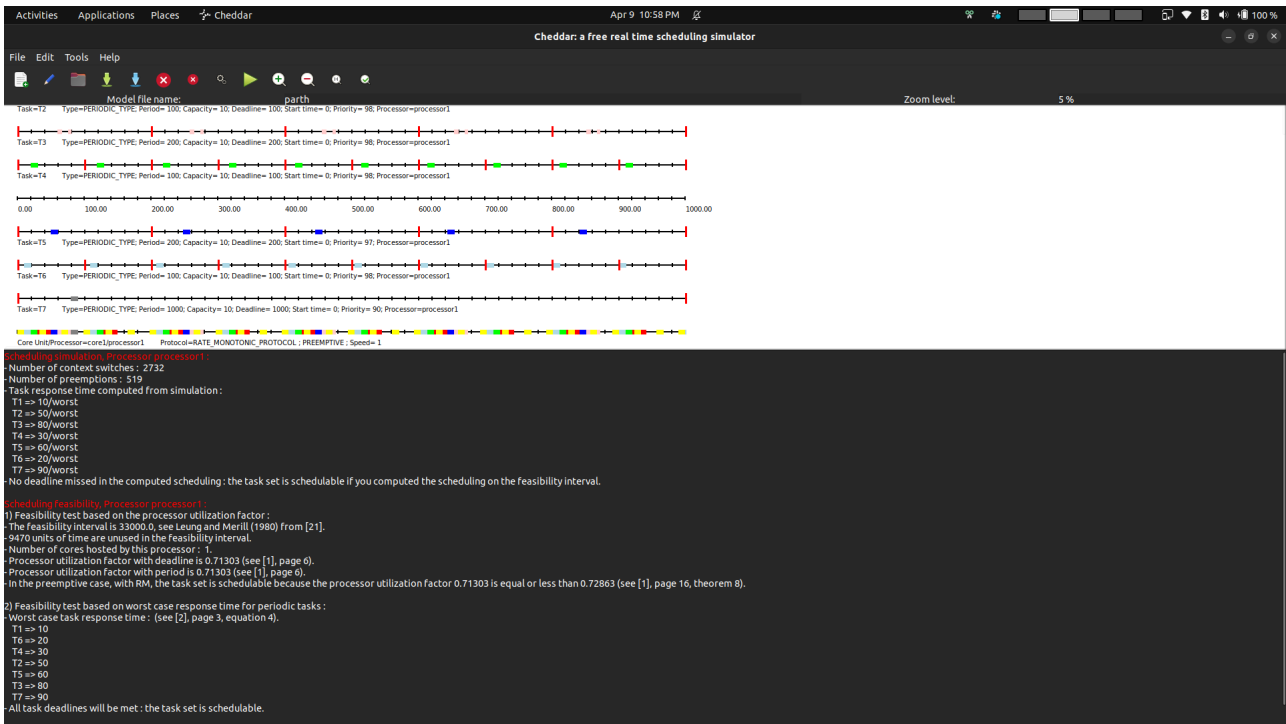


Figure 2: Cheddar Analysis for seqgen2 (300Hz)

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 $(\text{counter_isr} \% 10) == 0$.
 - Task 2, Task 4, and Task 6 are released every 30 cycles (990 ms) when $(\text{counter_isr} \% 30) == 0$.
 - Task 3 and Task 5 are released every 60 cycles (1980 ms) when $(\text{counter_isr} \% 60) == 0$.
 - Task 7 is released every 300 cycles (9900 ms) when $(\text{counter_isr} \% 300) == 0$.
- 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_exeectution_time 929 execution unit 90 *****
***** Task 2 wcet 11 total_exeectution_time 309 execution unit 30 *****
***** Task 3 wcet 10 total_exeectution_time 140 execution unit 14 *****
***** Task 4 wcet 10 total_exeectution_time 300 execution unit 30 *****
***** Task 5 wcet 11 total_exeectution_time 154 execution unit 14 *****
***** Task 6 wcet 10 total_exeectution_time 290 execution unit 29 *****
***** Task 7 wcet 10 total_exeectution_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 Time: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 | 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 3: C, T, D, for 10ms load, and sequencer frequency set to 30Hz

2.1.2 Cheddar

2.2 TIVA freeRTOS similar to seqgen2x(300Hz)

Output

```

***** Task 1 wcet 13 total_execlution_time 1072 execution unit 90 *****
***** Task 2 wcet 13 total_execlution_time 351 execution unit 29 *****
***** Task 3 wcet 13 total_execlution_time 176 execution unit 14 *****
***** Task 4 wcet 13 total_execlution_time 361 execution unit 29 *****
***** Task 5 wcet 12 total_execlution_time 168 execution unit 14 *****
***** Task 6 wcet 12 total_execlution_time 345 execution unit 29 *****
***** Task 7 wcet 12 total_execlution_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 4: C, T, D, for 10ms load, and sequencer frequency set to 30Hz

2.2.2 Cheddar

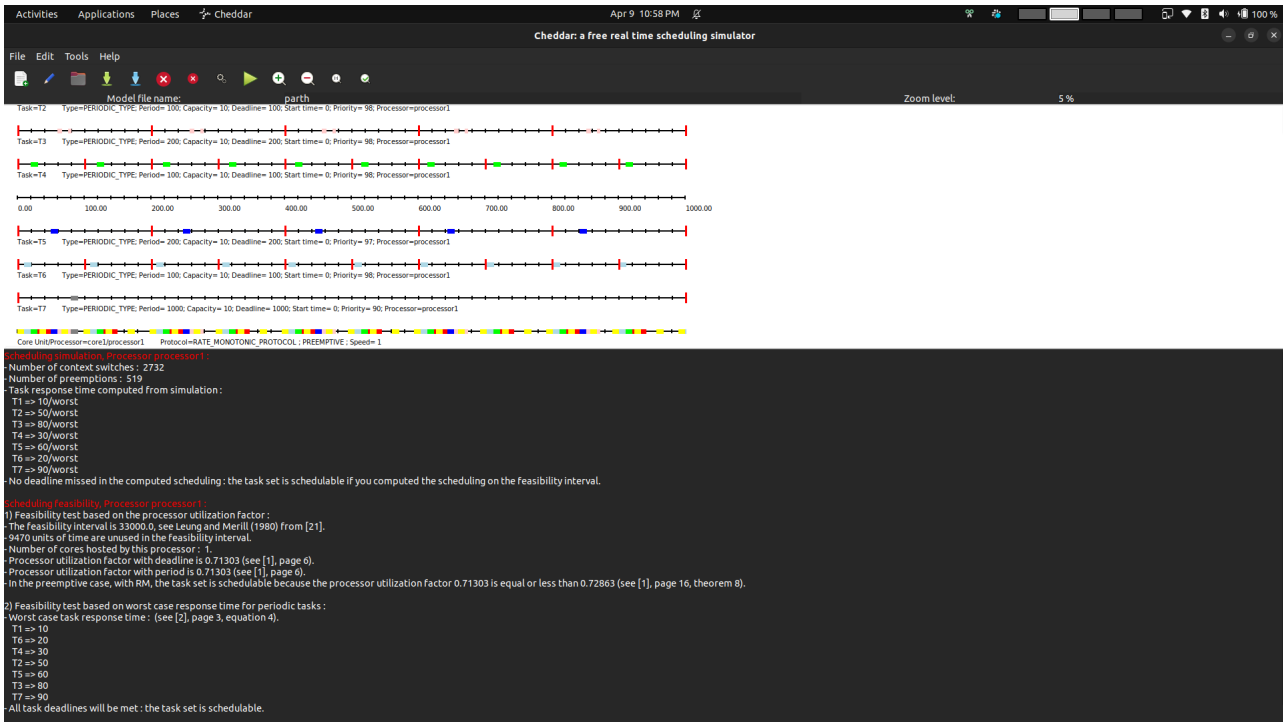


Figure 3: Cheddar Analysis for seqgen2 (300Hz)

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. Calculate the % 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 printf's 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_exeuction_time 9040 execution unit 9000 *****
***** Task 2 wcet 1 total_exeuction_time 2999 execution unit 2999 *****
***** Task 3 wcet 1 total_exeuction_time 1499 execution unit 1499 *****
***** Task 4 wcet 2 total_exeuction_time 4802 execution unit 2999 *****
```

```

***** Task 5 wcet 1 total_exeception_time 1499 execution unit 1499 *****
***** Task 6 wcet 2 total_exeception_time 3272 execution unit 2999 *****
***** Task 7 wcet 1 total_exeception_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 5: C, T, D, for 10ms load, and sequencer frequency set to 3000Hz

3.1.2 Cheddar

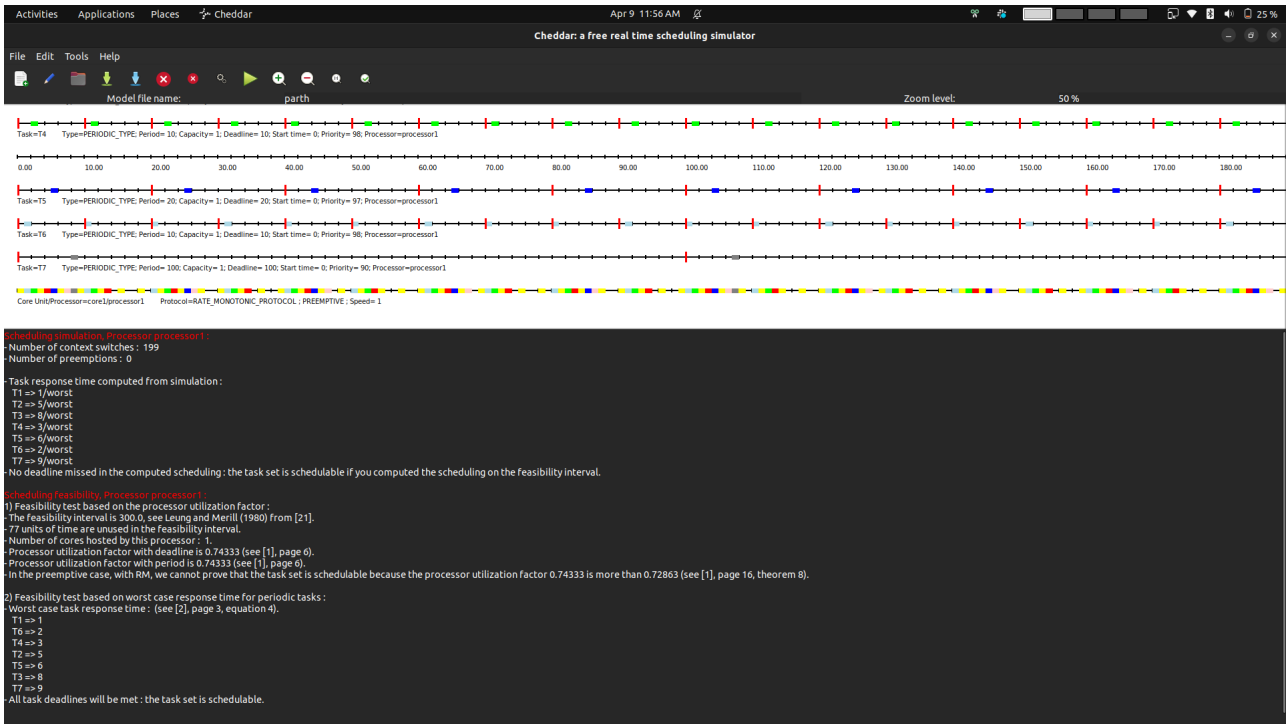


Figure 4: Cheddar Analysis for seqgen (30000Hz)

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

```
1
2 #include <stdint.h>
3 #include <stdbool.h>
4 #include "main.h"
5 #include "drivers/pinout.h"
6 #include "utils/uartstdio.h"
7
8 // TivaWare includes
9 #include "driverlib/sysctl.h"
10 #include "driverlib/debug.h"
11 #include "driverlib/rom_map.h"
12 #include "driverlib/rom.h"
13 #include "driverlib/timer.h"
14 #include "driverlib/inc/hw_memmap.h"
15 #include "driverlib/inc/hw_ints.h"
16
17 // FreeRTOS includes
18 #include "FreeRTOSConfig.h"
19 #include "FreeRTOS.h"
20 #include <timers.h>
21 #include <semphr.h>
22 #include "task.h"
23 #include "queue.h"
24 #include "limits.h"
25
26 #define FIB_LIMIT_FOR_32_BIT 47
27 #define ITERATION 120
28 #define MULTIPLIER 100
29 #define Hz (30 * MULTIPLIER) // Hz
30 #define SEQUENCER_COUNT (900 * MULTIPLIER)
31 #define UART_BAUD_RATE 1000000
32
33 SemaphoreHandle_t task_1_SyncSemaphore, task_2_SyncSemaphore, task_3_SyncSemaphore,
task_4_SyncSemaphore, task_5_SyncSemaphore, task_6_SyncSemaphore,
task_7_SyncSemaphore;
34 TickType_t startTimeTick;
35 TaskHandle_t Task1_handle, Task2_handle, Task3_handle, Task4_handle, Task5_handle,
Task6_handle, Task7_handle;
36 volatile uint32_t counter_isr = 0;
37 uint32_t ulPeriod;
38 volatile bool abort_test = false;
39 uint32_t wcet[7];
40 uint32_t execution_time[7];
41 uint32_t execution_cycle[7];
42
43 void init_Timer();
44 void init_Uart();
45 void init_Clock();
46
47 void fibonacci()
48 {
49     uint32_t i,j;
50     uint32_t fib = 1, fib_a = 1, fib_b = 1;
51     for ( i=0; i<ITERATION; i++)
```

```

52     {
53         for(j=0; j<FIB_LIMIT_FOR_32_BIT; j++){
54             fib_a = fib_b;
55             fib_b = fib;
56             fib = fib_a + fib_b;
57         }
58     }
59 }
60 }
61
62
63 void print_data(){
64     uint32_t i = 0;
65     for (i = 0; i < 7; i++){
66         UARTprintf("***** Task %d wcet %d total_executation_time %d execution unit %d
*****\n\r", i+1, wcet[i], execution_time[i], execution_cycle[i]);
67     }
68 }
69
70 void Timer0Isr_Sequencer(void)
71 {
72     TickType_t xCurrentTick = xTaskGetTickCount();
73     ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
74     counter_isr++;
75
76     // UARTprintf("Sequencer Thread ran at %d ms and Cycle of sequencer %d \n\r",
xCurrentTick, counter_isr);
77
78     if ((counter_isr % 10) == 0)
79     {
80         // Service_1 = RT_MAX-1 @ 300 Hz
81         xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
82     }
83
84     if ((counter_isr % 30) == 0)
85     {
86         // Service_2 = RT_MAX-2 @ 100 Hz
87         xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis
thread
88         // Service_4 = RT_MAX-2 @ 100 Hz
89         xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
90         // Service_6 = RT_MAX-2 @ 100 Hz
91         xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote
thread
92     }
93
94
95     if ((counter_isr % 60) == 0)
96     {
97         // Service_3 = RT_MAX-3 @ 50 Hz
98         xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
99         // Service_5 = RT_MAX-3 @ 50 Hz
100        xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
101    }
102
103
104    if ((counter_isr % 300) == 0)

```

```

105     {
106         // Service_7 = RT_MIN    10 Hz
107         xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
108     }
109
110     if (counter_isr > SEQUENCER_COUNT)
111     {
112         xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
113         xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis
114         thread
115         xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
116         xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
117         xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
118         xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote
119         thread
120         xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
121         abort_test = true;
122         ROM_TimerDisable(TIMER0_BASE, TIMER_A);
123         print_data();
124     }
125 }
126
127 // Process 1
128 void xTask1(void *pvParameters)
129 {
130     BaseType_t xResult;
131
132     while (!abort_test)
133     {
134         xResult = xSemaphoreTake(task_1_SyncSemaphore, portMAX_DELAY);
135
136         if (xResult == pdPASS)
137         {
138             execution_cycle[0]++;
139             TickType_t xCurrentTick = xTaskGetTickCount();
140             UARTprintf("T1 S:%d, R %d\n\r", xCurrentTick, execution_cycle[0]);
141             fibonacci();
142             TickType_t xFibTime = xTaskGetTickCount();
143             TickType_t total_time = (xFibTime - xCurrentTick);
144             execution_time[0] += total_time;
145             if(wcet[0] < total_time) wcet[0] = total_time;
146
147             UARTprintf("T1 C:%d, E:%d\n\r", xFibTime, total_time);
148         }
149     }
150     vTaskDelete( NULL );
151 }
152
153 void xTask2(void *pvParameters)
154 {
155     BaseType_t xResult;
156
157     while (!abort_test)
158     {
159         xResult = xSemaphoreTake(task_2_SyncSemaphore, portMAX_DELAY);
160     }
161 }

```

```
160
161     if (xResult == pdPASS)
162     {
163         execution_cycle[1]++;
164         TickType_t xCurrentTick = xTaskGetTickCount();
165         UARTprintf("T2 S:%d, R %d\n\r", xCurrentTick, execution_cycle[1]);
166         fibonacci();
167         TickType_t xFibTime = xTaskGetTickCount();
168         TickType_t total_time = (xFibTime - xCurrentTick);
169         execution_time[1] += total_time;
170         if(wcet[1] < total_time) wcet[1] = total_time;
171         UARTprintf("T2 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
172     }
173 }
174 vTaskSuspend( NULL );
175 }
176 void xTask3(void *pvParameters)
177 {
178     BaseType_t xResult;;
179
180     while (!abort_test)
181     {
182
183         xResult = xSemaphoreTake(task_3_SyncSemaphore, portMAX_DELAY);
184
185         if (xResult == pdPASS)
186         {
187             execution_cycle[2]++;
188             TickType_t xCurrentTick = xTaskGetTickCount();
189             UARTprintf("T3 S:%d, R %d\n\r", xCurrentTick, execution_cycle[2]);
190             fibonacci();
191             TickType_t xFibTime = xTaskGetTickCount();
192             TickType_t total_time = (xFibTime - xCurrentTick);
193             execution_time[2] += total_time;
194             if(wcet[2] < total_time) wcet[2] = total_time;
195             UARTprintf("T3 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
196         }
197     }
198     vTaskDelete( NULL );
199 }
200 void xTask4(void *pvParameters)
201 {
202     BaseType_t xResult;
203
204     while (!abort_test)
205     {
206
207         xResult = xSemaphoreTake(task_4_SyncSemaphore, portMAX_DELAY);
208
209         if (xResult == pdPASS)
210         {
211             execution_cycle[3]++;
212             TickType_t xCurrentTick = xTaskGetTickCount();
213             UARTprintf("T4 S:%d, R %d\n\r", xCurrentTick, execution_cycle[3]);
214             fibonacci();
215             TickType_t xFibTime = xTaskGetTickCount();
```

```
216         TickType_t total_time = (xFibTime - xCurrentTick);
217         execution_time[3] += total_time;
218         if(wcet[3] < total_time) wcet[3] = total_time;
219         UARTprintf("T4 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
220     }
221 }
222 vTaskDelete( NULL );
223 }
224 void xTask5(void *pvParameters)
225 {
226     BaseType_t xResult;
227
228     while (!abort_test)
229     {
230
231         xResult = xSemaphoreTake(task_5_SyncSemaphore, portMAX_DELAY);
232
233         if (xResult == pdPASS)
234         {
235             execution_cycle[4]++;
236             TickType_t xCurrentTick = xTaskGetTickCount();
237             UARTprintf("T5 S:%d, R %d\n\r", xCurrentTick, execution_cycle[4]);
238             fibonacci();
239             TickType_t xFibTime = xTaskGetTickCount();
240             TickType_t total_time = (xFibTime - xCurrentTick);
241             execution_time[4] += total_time;
242             if(wcet[4] < total_time) wcet[4] = total_time;
243             UARTprintf("T5 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
244         }
245     }
246     vTaskDelete( NULL );
247 }
248 void xTask6(void *pvParameters)
249 {
250     BaseType_t xResult;
251
252     while (!abort_test)
253     {
254
255         xResult = xSemaphoreTake(task_6_SyncSemaphore, portMAX_DELAY);
256
257         if (xResult == pdPASS)
258         {
259             execution_cycle[5]++;
260             TickType_t xCurrentTick = xTaskGetTickCount();
261             UARTprintf("T6 S:%d, R %d\n\r", xCurrentTick, execution_cycle[5]);
262             fibonacci();
263             TickType_t xFibTime = xTaskGetTickCount();
264             TickType_t total_time = (xFibTime - xCurrentTick);
265             execution_time[5] += total_time;
266             if(wcet[5] < total_time) wcet[5] = total_time;
267             UARTprintf("T6 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
268         }
269     }
270     vTaskDelete( NULL );
271 }
```

```
272 void xTask7(void *pvParameters)
273 {
274     BaseType_t xResult;
275
276     while (!abort_test)
277     {
278         xResult = xSemaphoreTake(task_7_SyncSemaphore, portMAX_DELAY);
279
280         if (xResult == pdPASS)
281         {
282             execution_cycle[6]++;
283             TickType_t xCurrentTick = xTaskGetTickCount();
284             UARTprintf("T7 S:%d , R %d\n\r", xCurrentTick, execution_cycle[6]);
285             fibonacci();
286             TickType_t xFibTime = xTaskGetTickCount();
287             TickType_t total_time = (xFibTime - xCurrentTick);
288             execution_time[6] += total_time;
289             if(wcet[6] < total_time) wcet[6] = total_time;
290             UARTprintf("T7 C:%d, E:%d\n\r", xFibTime, (xFibTime - xCurrentTick));
291         }
292     }
293     vTaskDelete( NULL );
294 }
295
296 // Main function
297 int main(void)
298 {
299     init_Clock();
300     init_Uart();
301     init_Timer();
302
303     task_1_SyncSemaphore = xSemaphoreCreateBinary();
304     task_2_SyncSemaphore = xSemaphoreCreateBinary();
305     task_3_SyncSemaphore = xSemaphoreCreateBinary();
306     task_4_SyncSemaphore = xSemaphoreCreateBinary();
307     task_5_SyncSemaphore = xSemaphoreCreateBinary();
308     task_6_SyncSemaphore = xSemaphoreCreateBinary();
309     task_7_SyncSemaphore = xSemaphoreCreateBinary();
310
311     UARTprintf("Cyclic executer : %d Hz\n\r", Hz);
312     xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 4, &Task1_handle);
313     xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 3, &Task2_handle);
314     xTaskCreate(xTask3, "Task3", configMINIMAL_STACK_SIZE, NULL, 2, &Task3_handle);
315     xTaskCreate(xTask4, "Task4", configMINIMAL_STACK_SIZE, NULL, 3, &Task4_handle);
316     xTaskCreate(xTask5, "Task5", configMINIMAL_STACK_SIZE, NULL, 2, &Task5_handle);
317     xTaskCreate(xTask6, "Task6", configMINIMAL_STACK_SIZE, NULL, 3, &Task6_handle);
318     xTaskCreate(xTask7, "Task7", configMINIMAL_STACK_SIZE, NULL, 1, &Task7_handle);
319
320     startTimeTick = xTaskGetTickCount();
321
322     vTaskStartScheduler();
323     UARTprintf("\nTEST COMPLETE\n");
324     return (0);
325 }
326
327
```

```
328 void init_Timer()  
329 {  
330     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);  
331     ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);           // 32 bits Timer  
332     TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr_Sequencer); // Registering isr  
333  
334     ulPeriod = (SYSTEM_CLOCK / Hz);  
335     ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 1);  
336  
337     ROM_TimerEnable(TIMER0_BASE, TIMER_A);  
338     ROM_IntEnable(INT_TIMER0A);  
339     ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);  
340 }  
341  
342 void init_Clock()  
343 {  
344     // Initialize system clock to 120 MHz  
345     uint32_t output_clock_rate_hz;  
346     output_clock_rate_hz = ROM_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |  
SYSCTL_OSC_MAIN | SYSCTL_USE_PCL | SYSCTL_CFG_VCO_480), SYSTEM_CLOCK);  
347     ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);  
348 }  
349  
350 void init_Uart()  
351 {  
352     // Initialize the GPIO pins for the Launchpad  
353     PinoutSet(false, false);  
354     UARTStdioConfig(0, UART_BAUD_RATE, SYSTEM_CLOCK);  
355 }  
356  
357 /* ASSERT() Error function  
358 *  
359 * failed ASSERTS() from driverlib/debug.h are executed in this function  
360 */  
361 void __error__(char *pcFilename, uint32_t ui32Line)  
362 {  
363     // Place a breakpoint here to capture errors until logging routine is finished  
364     while (1)  
365     {  
366     }  
367 }  
368
```


seqgen.c

```

1  /* ===== */
2  /* */
3  // Sam Siewert, December 2017
4  //
5  // Sequencer Generic
6  //
7  // The purpose of this code is to provide an example for how to best
8  // sequence a set of periodic services for problems similar to and including
9  // the final project in real-time systems.
10 //
11 // For example: Service_1 for camera frame acquisition
12 //               Service_2 for image analysis and timestamping
13 //               Service_3 for image processing (difference images)
14 //               Service_4 for save time-stamped image to file service
15 //               Service_5 for save processed image to file service
16 //               Service_6 for send image to remote server to save copy
17 //               Service_7 for elapsed time in syslog each minute for debug
18 //
19 // At least two of the services need to be real-time and need to run on a single
20 // core or run without affinity on the SMP cores available to the Linux
21 // scheduler as a group. All services can be real-time, but you could choose
22 // to make just the first 2 real-time and the others best effort.
23 //
24 // For the standard project, to time-stamp images at the 1 Hz rate with unique
25 // clock images (unique second hand / seconds) per image, you might use the
26 // following rates for each service:
27 //
28 // Sequencer - 30 Hz
29 //               [gives semaphores to all other services]
30 // Service_1 - 3 Hz , every 10th Sequencer loop
31 //               [buffers 3 images per second]
32 // Service_2 - 1 Hz , every 30th Sequencer loop
33 //               [time-stamp middle sample image with cvPutText or header]
34 // Service_3 - 0.5 Hz, every 60th Sequencer loop
35 //               [difference current and previous time stamped images]
36 // Service_4 - 1 Hz, every 30th Sequencer loop
37 //               [save time stamped image with cvSaveImage or write()]
38 // Service_5 - 0.5 Hz, every 60th Sequencer loop
39 //               [save difference image with cvSaveImage or write()]
40 // Service_6 - 1 Hz, every 30th Sequencer loop
41 //               [write current time-stamped image to TCP socket server]
42 // Service_7 - 0.1 Hz, every 300th Sequencer loop
43 //               [syslog the time for debug]
44 //
45 // With the above, priorities by RM policy would be:
46 //
47 // Sequencer = RT_MAX @ 30 Hz
48 // Service_1 = RT_MAX-1 @ 3 Hz
49 // Service_2 = RT_MAX-2 @ 1 Hz
50 // Service_3 = RT_MAX-3 @ 0.5 Hz
51 // Service_4 = RT_MAX-2 @ 1 Hz
52 // Service_5 = RT_MAX-3 @ 0.5 Hz
53 // Service_6 = RT_MAX-2 @ 1 Hz

```

```
54 // Service_7 = RT_MIN    0.1 Hz
55 //
56 // Here are a few hardware/platform configuration settings on your Jetson
57 // that you should also check before running this code:
58 //
59 // 1) Check to ensure all your CPU cores on in an online state.
60 //
61 // 2) Check /sys/devices/system/cpu or do lscpu.
62 //
63 // Tegra is normally configured to hot-plug CPU cores, so to make all
64 // available, as root do:
65 //
66 // echo 0 > /sys/devices/system/cpu/cpuquiet/tegra_cpuquiet/enable
67 // echo 1 > /sys/devices/system/cpu/cpu1/online
68 // echo 1 > /sys/devices/system/cpu/cpu2/online
69 // echo 1 > /sys/devices/system/cpu/cpu3/online
70 //
71 // 3) Check for precision time resolution and support with cat /proc/timer_list
72 //
73 // 4) Ideally all printf calls should be eliminated as they can interfere with
74 // timing. They should be replaced with an in-memory event logger or at
75 // least calls to syslog.
76 //
77 // 5) For simplicity, you can just allow Linux to dynamically load balance
78 // threads to CPU cores (not set affinity) and as long as you have more
79 // threads than you have cores, this is still an over-subscribed system
80 // where RM policy is required over the set of cores.
81
82 // This is necessary for CPU affinity macros in Linux
83 #define _GNU_SOURCE
84
85 #include <stdio.h>
86 #include <stdlib.h>
87 #include <unistd.h>
88
89 #include <pthread.h>
90 #include <sched.h>
91 #include <time.h>
92 #include <semaphore.h>
93
94 #include <syslog.h>
95 #include <sys/time.h>
96
97 #include <errno.h>
98
99 #define USEC_PER_MSEC (1000)
100 #define MS_PER_SEC (1000)
101 #define NANOSEC_PER_SEC (1000000000)
102 #define NUM_CPU_CORES (1)
103 #define TRUE (1)
104 #define FALSE (0)
105 #define ITERATION_COUNT 466500 // 100 ms load
106
107 #define NUM_THREADS (7 + 1)
108
109 int abortTest = FALSE;
```

```

110 int abortS1 = FALSE, abortS2 = FALSE, abortS3 = FALSE, abortS4 = FALSE, abortS5 =
    FALSE, abortS6 = FALSE, abortS7 = FALSE;
111 sem_t semS1, semS2, semS3, semS4, semS5, semS6, semS7;
112 struct timeval start_time_val;
113
114 double wcet[7];
115 double execution_time[7];
116 int execution_cycle[7];
117
118 typedef struct
119 {
120     int threadIdx;
121     unsigned long long sequencePeriods;
122 } threadParams_t;
123
124 void *Sequencer(void *threadp);
125
126 void *Service_1(void *threadp);
127 void *Service_2(void *threadp);
128 void *Service_3(void *threadp);
129 void *Service_4(void *threadp);
130 void *Service_5(void *threadp);
131 void *Service_6(void *threadp);
132 void *Service_7(void *threadp);
133 double getTimeMsec(void);
134 void print_scheduler(void);
135
136 #define FIB_LIMIT_FOR_32_BIT 47
137 #define ITERATION_COUNT_FIB 15000
138
139 void fibTest(int interation_count)
140 {
141     int fib, fib0, fib1;
142     int jdx = 0;
143     for (int idx = 0; idx < interation_count; idx++)
144     {
145         fib = fib0 + fib1;
146         while (jdx < FIB_LIMIT_FOR_32_BIT)
147         {
148             fib0 = fib1;
149             fib1 = fib;
150             fib = fib0 + fib1;
151             jdx++;
152         }
153         jdx = 0;
154     }
155 }
156
157
158 void print_data(){
159     for(int i=0; i<7; i++){
160         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]);
161         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]);

```

```

162     }
163
164 }
165
166 double read_time(double *var)
167 {
168     struct timeval tv;
169     if (gettimeofday(&tv, NULL) != 0)
170     {
171         perror("readTOD");
172         return 0.0;
173     }
174     else
175     {
176         *var = ((double)((((double)tv.tv_sec * 1000) + (((double)tv.tv_usec) / 1000.0)
177     ));
178     }
179     return (*var);
180 }
181
182 void main(void)
183 {
184     struct timeval current_time_val;
185     int i, rc, scope;
186     cpu_set_t threadcpu;
187     pthread_t threads[NUM_THREADS];
188     threadParams_t threadParams[NUM_THREADS];
189     pthread_attr_t rt_sched_attr[NUM_THREADS];
190     int rt_max_prio, rt_min_prio;
191     struct sched_param rt_param[NUM_THREADS];
192     struct sched_param main_param;
193     pthread_attr_t main_attr;
194     pid_t mainpid;
195     cpu_set_t allcpuset;
196
197     printf("Starting Sequencer Demo\n");
198     syslog(LOG_CRIT, "Starting Sequencer Demo\n");
199
200     printf("testing Fib load with iterations :%d\n", ITERATION_COUNT);
201     double avg_time = 0;
202     for(int i=0;i<10;i++){
203         double start, end;
204         read_time(&start);
205         fibTest(ITERATION_COUNT);
206         read_time(&end);
207         double total_ex = end - start;
208         avg_time += total_ex;
209         printf("iteration %d) Start time: %f ms , end time: %f ms , execution time:
%f ms\n\n",i, start, end, total_ex);
210         syslog(LOG_CRIT, "iteration %d) Start time: %f ms , end time: %f ms ,
execution time: %f ms\n\n",i, start, end, total_ex);
211     }
212
213     printf("***** Average time %f *****\n", avg_time / 10);
214     syslog(LOG_CRIT, "***** Average time %f *****\n", avg_time / 10);
215

```

```
216
217     gettimeofday(&start_time_val, (struct timezone *)0);
218     gettimeofday(&current_time_val, (struct timezone *)0);
219     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);
220     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);
221
222     printf("System has %d processors configured and %d available.\n",
get_nprocs_conf(), get_nprocs());
223     syslog(LOG_CRIT, "System has %d processors configured and %d available.\n",
get_nprocs_conf(), get_nprocs());
224
225     CPU_ZERO(&allcpuset);
226
227     for (i = 0; i < NUM_CPU_CORES; i++)
228         CPU_SET(i, &allcpuset);
229
230     printf("Using CPUS=%d from total available.\n", CPU_COUNT(&allcpuset));
231
232     // initialize the sequencer semaphores
233     //
234     if (sem_init(&semS1, 0, 0))
235     {
236         printf("Failed to initialize S1 semaphore\n");
237         exit(-1);
238     }
239     if (sem_init(&semS2, 0, 0))
240     {
241         printf("Failed to initialize S2 semaphore\n");
242         exit(-1);
243     }
244     if (sem_init(&semS3, 0, 0))
245     {
246         printf("Failed to initialize S3 semaphore\n");
247         exit(-1);
248     }
249     if (sem_init(&semS4, 0, 0))
250     {
251         printf("Failed to initialize S4 semaphore\n");
252         exit(-1);
253     }
254     if (sem_init(&semS5, 0, 0))
255     {
256         printf("Failed to initialize S5 semaphore\n");
257         exit(-1);
258     }
259     if (sem_init(&semS6, 0, 0))
260     {
261         printf("Failed to initialize S6 semaphore\n");
262         exit(-1);
263     }
264     if (sem_init(&semS7, 0, 0))
265     {
266         printf("Failed to initialize S7 semaphore\n");
267         exit(-1);
268     }
```

```

269
270     mainpid = getpid();
271
272     rt_max_prio = sched_get_priority_max(SCHED_FIFO);
273     rt_min_prio = sched_get_priority_min(SCHED_FIFO);
274
275     rc = sched_getparam(mainpid, &main_param);
276     main_param.sched_priority = rt_max_prio;
277     rc = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
278     if (rc < 0)
279         perror("main_param");
280     print_scheduler();
281
282     pthread_attr_getscope(&main_attr, &scope);
283
284     if (scope == PTHREAD_SCOPE_SYSTEM)
285         printf("PTHREAD SCOPE SYSTEM\n");
286     else if (scope == PTHREAD_SCOPE_PROCESS)
287         printf("PTHREAD SCOPE PROCESS\n");
288     else
289         printf("PTHREAD SCOPE UNKNOWN\n");
290
291     printf("rt_max_prio=%d\n", rt_max_prio);
292     printf("rt_min_prio=%d\n", rt_min_prio);
293
294     for (i = 0; i < NUM_THREADS; i++)
295     {
296
297         CPU_ZERO(&threadcpu);
298         CPU_SET(3, &threadcpu);
299
300         rc = pthread_attr_init(&rt_sched_attr[i]);
301         rc = pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
302         rc = pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
303         rc = pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &
threadcpu);
304
305         rt_param[i].sched_priority = rt_max_prio - i;
306         pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);
307
308         threadParams[i].threadIdx = i;
309     }
310
311     printf("Service threads will run on %d CPU cores\n", CPU_COUNT(&threadcpu));
312     syslog(LOG_CRIT, "Service threads will run on %d CPU cores\n", CPU_COUNT(&
threadcpu));
313
314     // Create Service threads which will block awaiting release for:
315     //
316
317     // Servcie_1 = RT_MAX-1 @ 3 Hz
318     //
319     rt_param[1].sched_priority = rt_max_prio - 1;
320     pthread_attr_setschedparam(&rt_sched_attr[1], &rt_param[1]);
321     rc = pthread_create(&threads[1], // pointer to thread descriptor
322                        &rt_sched_attr[1], // use specific attributes
323                        //(void *)0, // default attributes

```

```
324         Service_1,                // thread function entry point
325         (void *)&(threadParams[1]) // parameters to pass in
326     );
327     if (rc < 0)
328         perror("pthread_create for service 1");
329     else{
330         printf("pthread_create successful for service 1\n");
331         syslog(LOG_CRIT, "pthread_create successful for service 1\n");
332     }
333
334     // Service_2 = RT_MAX-2 @ 1 Hz
335     //
336     rt_param[2].sched_priority = rt_max_prio - 2;
337     pthread_attr_setschedparam(&rt_sched_attr[2], &rt_param[2]);
338     rc = pthread_create(&threads[2], &rt_sched_attr[2], Service_2, (void *)&
339 (threadParams[2]));
340     if (rc < 0)
341         perror("pthread_create for service 2");
342     else{
343         printf("pthread_create successful for service 2\n");
344         syslog(LOG_CRIT, "pthread_create successful for service 2\n");
345     }
346
347     // Service_3 = RT_MAX-3 @ 0.5 Hz
348     //
349     rt_param[3].sched_priority = rt_max_prio - 3;
350     pthread_attr_setschedparam(&rt_sched_attr[3], &rt_param[3]);
351     rc = pthread_create(&threads[3], &rt_sched_attr[3], Service_3, (void *)&
352 (threadParams[3]));
353     if (rc < 0)
354         perror("pthread_create for service 3");
355     else{
356         printf("pthread_create successful for service 3\n");
357         syslog(LOG_CRIT, "pthread_create successful for service 3\n");
358     }
359
360     // Service_4 = RT_MAX-2 @ 1 Hz
361     //
362     rt_param[4].sched_priority = rt_max_prio - 2;
363     pthread_attr_setschedparam(&rt_sched_attr[4], &rt_param[4]);
364     rc = pthread_create(&threads[4], &rt_sched_attr[4], Service_4, (void *)&
365 (threadParams[4]));
366     if (rc < 0)
367         perror("pthread_create for service 4");
368     else{
369         printf("pthread_create successful for service 4\n");
370         syslog(LOG_CRIT, "pthread_create successful for service 4\n");
371     }
372
373     // Service_5 = RT_MAX-3 @ 0.5 Hz
374     //
375     rt_param[5].sched_priority = rt_max_prio - 3;
376     pthread_attr_setschedparam(&rt_sched_attr[5], &rt_param[5]);
377     rc = pthread_create(&threads[5], &rt_sched_attr[5], Service_5, (void *)&
378 (threadParams[5]));
379     if (rc < 0)
380         perror("pthread_create for service 5");
```

```

377     else{
378
379         printf("pthread_create successful for service 5\n");
380         syslog(LOG_CRIT, "pthread_create successful for service 5\n");
381     }
382
383     // Service_6 = RT_MAX-2 @ 1 Hz
384     //
385     rt_param[6].sched_priority = rt_max_prio - 2;
386     pthread_attr_setschedparam(&rt_sched_attr[6], &rt_param[6]);
387     rc = pthread_create(&threads[6], &rt_sched_attr[6], Service_6, (void *)&
(threadParams[6]));
388     if (rc < 0)
389         perror("pthread_create for service 6");
390     else{
391
392         syslog(LOG_CRIT, "pthread_create successful for service 6\n");
393     }
394
395     // Service_7 = RT_MIN    0.1 Hz
396     //
397     rt_param[7].sched_priority = rt_min_prio;
398     pthread_attr_setschedparam(&rt_sched_attr[7], &rt_param[7]);
399     rc = pthread_create(&threads[7], &rt_sched_attr[7], Service_7, (void *)&
(threadParams[7]));
400     if (rc < 0)
401         perror("pthread_create for service 7");
402     else{
403
404         printf("pthread_create successful for service 7\n");
405         syslog(LOG_CRIT, "pthread_create successful for service 7\n");
406     }
407
408     // Wait for service threads to initialize and await release by sequencer.
409     //
410     // Note that the sleep is not necessary of RT service threads are created with
411     // correct POSIX SCHED_FIFO priorities compared to non-RT priority of this main
412     // program.
413     //
414     // usleep(1000000);
415
416     // Create Sequencer thread, which like a cyclic executive, is highest prio
417     printf("Start sequencer\n");
418     syslog(LOG_CRIT, "Start sequencer\n");
419     threadParams[0].sequencePeriods = 900;
420
421     // Sequencer = RT_MAX    @ 30 Hz
422     //
423     rt_param[0].sched_priority = rt_max_prio;
424     pthread_attr_setschedparam(&rt_sched_attr[0], &rt_param[0]);
425     rc = pthread_create(&threads[0], &rt_sched_attr[0], Sequencer, (void *)&
(threadParams[0]));
426     if (rc < 0)
427         perror("pthread_create for sequencer service 0");
428     else{
429
430         printf("pthread_create successful for sequencer service 0\n");

```



```

431     syslog(LOG_CRIT, "pthread_create successful for sequencer service 0\n");
432 }
433
434 for (i = 0; i < NUM_THREADS; i++)
435     pthread_join(threads[i], NULL);
436
437 printf("\nTEST COMPLETE\n");
438 syslog(LOG_CRIT, "\nTEST COMPLETE\n");
439 }
440
441 void *Sequencer(void *threadp)
442 {
443     struct timeval current_time_val;
444     struct timespec delay_time = {0, 33333333}; // delay for 33.33 msec, 30 Hz
445     struct timespec remaining_time;
446     double current_time;
447     double residual;
448     int rc, delay_cnt = 0;
449     unsigned long long seqCnt = 0;
450     threadParams_t *threadParams = (threadParams_t *)threadp;
451
452     gettimeofday(&current_time_val, (struct timezone *)0);
453     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);
454     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);
455
456     do
457     {
458         delay_cnt = 0;
459         residual = 0.0;
460
461         gettimeofday(&current_time_val, (struct timezone *)0);
462         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);
463
464         do
465         {
466             rc = nanosleep(&delay_time, &remaining_time);
467
468             if (rc == EINTR)
469             {
470                 residual = remaining_time.tv_sec + ((double)remaining_time.tv_nsec /
(double)NANOSEC_PER_SEC);
471
472                 if (residual > 0.0)
473                     printf("residual=%lf, sec=%d, nsec=%d\n", residual, (int)
remaining_time.tv_sec, (int)remaining_time.tv_nsec);
474
475                 delay_cnt++;
476             }
477             else if (rc < 0)
478             {
479                 perror("Sequencer nanosleep");
480                 exit(-1);
481             }

```

```

482
483     } while ((residual > 0.0) && (delay_cnt < 100));
484
485     seqCnt++;
486     gettimeofday(&current_time_val, (struct timezone *)0);
487     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);
488
489     if (delay_cnt > 1)
490         printf("Sequencer looping delay %d\n", delay_cnt);
491
492     // Release each service at a sub-rate of the generic sequencer rate
493
494     // Service_1 = RT_MAX-1 @ 3 Hz
495     if ((seqCnt % 10) == 0)
496     {
497         syslog(LOG_CRIT, "Task 1 (Frame Sampler thread) Released \n");
498         sem_post(&semS1); // Frame Sampler thread
499     }
500
501     // Service_2 = RT_MAX-2 @ 1 Hz
502     if ((seqCnt % 30) == 0)
503     {
504         syslog(LOG_CRIT, "Task 2 (Time-stamp with Image Analysis thread) Released
\n");
505         sem_post(&semS2); // Time-stamp with Image Analysis thread
506     }
507
508     // Service_3 = RT_MAX-3 @ 0.5 Hz
509     if ((seqCnt % 60) == 0)
510     {
511         syslog(LOG_CRIT, "Task 3 ( Difference Image Proc thread) Released \n");
512         sem_post(&semS3); // Difference Image Proc thread
513     }
514
515     // Service_4 = RT_MAX-2 @ 1 Hz
516     if ((seqCnt % 30) == 0)
517     {
518         syslog(LOG_CRIT, "Task 4 (Time-stamp Image Save to File thread) Released
\n");
519         sem_post(&semS4); // Time-stamp Image Save to File thread
520     }
521
522     // Service_5 = RT_MAX-3 @ 0.5 Hz
523     if ((seqCnt % 60) == 0)
524     {
525         syslog(LOG_CRIT, "Task 5 (Processed Image Save to File thread) Released
\n");
526         sem_post(&semS5); // Processed Image Save to File thread
527     }
528
529     // Service_6 = RT_MAX-2 @ 1 Hz
530     if ((seqCnt % 30) == 0)
531     {
532         syslog(LOG_CRIT, "Task 6 (Send Time-stamped Image to Remote thread)
Released \n");

```

```

533     sem_post(&semS6); // Send Time-stamped Image to Remote thread
534 }
535
536 // Service_7 = RT_MIN    0.1 Hz
537 if ((seqCnt % 300) == 0)
538 {
539     syslog(LOG_CRIT, "Task 7 (10 sec Tick Debug thread) Released \n");
540     sem_post(&semS7); // 10 sec Tick Debug thread
541 }
542
543 gettimeofday(&current_time_val, NULL);
544 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);
545
546 } while (!abortTest && (seqCnt < threadParams->sequencePeriods));
547
548 sem_post(&semS1);
549 sem_post(&semS2);
550 sem_post(&semS3);
551 sem_post(&semS4);
552 sem_post(&semS5);
553 sem_post(&semS6);
554 sem_post(&semS7);
555 abortS1 = TRUE;
556 abortS2 = TRUE;
557 abortS3 = TRUE;
558 abortS4 = TRUE;
559 abortS5 = TRUE;
560 abortS6 = TRUE;
561 abortS7 = TRUE;
562 print_data();
563
564 pthread_exit((void *)0);
565 }
566
567 void *Service_1(void *threadp)
568 {
569     double start, end, total;
570     threadParams_t *threadParams = (threadParams_t *)threadp;
571
572     read_time(&start);
573     syslog(LOG_CRIT, "Task 1, Frame Sampler thread @ msec=%f \n", start);
574     printf("Task 1, Frame Sampler thread @ msec=%f \n", start);
575
576     while (!abortS1)
577     {
578         sem_wait(&semS1);
579
580         execution_cycle[0]++;
581         read_time(&start);
582         syslog(LOG_CRIT, "Task 1, Frame Sampler start %d @ msec=%f",
execution_cycle[0], start);
583         fibTest(ITERATION_COUNT);
584         read_time(&end);
585         total = end - start;
586         if(total > wcet[0]) wcet[0] = total;

```

```
587     execution_time[0] += total;
588     syslog(LOG_CRIT, "Task 1, Frame Sampler Execution complete @ msec=%f,
execution time : %f ms\n", end, total);
589 }
590
591 pthread_exit((void *)0);
592 }
593
594 void *Service_2(void *threadp)
595 {
596
597     double start, end, total;
598     threadParams_t *threadParams = (threadParams_t *)threadp;
599
600     read_time(&start);
601     syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread @ msec=%f \n",
start);
602     printf("Task 2, Time-stamp with Image Analysis thread @ msec=%f \n", start);
603
604     while (!abortS2)
605     {
606         sem_wait(&semS2);
607
608         execution_cycle[1]++;
609         read_time(&start);
610         syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread start %d @
msec=%f", execution_cycle[1], start);
611         fibTest(ITERATION_COUNT);
612         read_time(&end);
613         total = end - start;
614         if(total > wcet[1]) wcet[1] = total;
615         execution_time[1] += total;
616         syslog(LOG_CRIT, "Task 2, Time-stamp with Image Analysis thread Execution
complete @ msec=%f, execution time : %f ms\n", end, total);
617     }
618
619     pthread_exit((void *)0);
620
621 }
622
623 void *Service_3(void *threadp)
624 {
625
626     double start, end, total;
627     threadParams_t *threadParams = (threadParams_t *)threadp;
628
629     read_time(&start);
630     syslog(LOG_CRIT, "Task 3, Difference Image Proc thread @ msec=%f \n", start);
631     printf("Task 3, Difference Image Proc thread @ msec=%f \n", start);
632
633     while (!abortS3)
634     {
635         sem_wait(&semS3);
636
637         execution_cycle[2]++;
638         read_time(&start);
```

```
639     syslog(LOG_CRIT, "Task 3, Difference Image Proc  start %d @ msec=%f",
execution_cycle[2], start);
640     fibTest(ITERATION_COUNT);
641     read_time(&end);
642     total = end - start;
643     if(total > wcet[2]) wcet[2] = total;
644     execution_time[2] += total;
645     syslog(LOG_CRIT, "Task 3, Difference Image Proc Execution complete @ msec=%f,
execution time : %f ms\n", end, total);
646 }
647
648 pthread_exit((void *)0);
649 }
650
651 void *Service_4(void *threadp)
652 {
653
654     double start, end, total;
655     threadParams_t *threadParams = (threadParams_t *)threadp;
656
657     read_time(&start);
658     syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File thread @ msec=%f \n",
start);
659     printf("Task 4, Time-stamp Image Save to File thread @ msec=%f \n", start);
660
661     while (!abortS4)
662     {
663         sem_wait(&semS4);
664
665         execution_cycle[3]++;
666         read_time(&start);
667         syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File start %d @ msec=%f",
execution_cycle[3], start);
668         fibTest(ITERATION_COUNT);
669         read_time(&end);
670         total = end - start;
671         if(total > wcet[3]) wcet[3] = total;
672         execution_time[3] += total;
673         syslog(LOG_CRIT, "Task 4, Time-stamp Image Save to File Execution complete @
msec=%f, execution time : %f ms\n", end, total);
674     }
675
676     pthread_exit((void *)0);
677 }
678
679 void *Service_5(void *threadp)
680 {
681
682     double start, end, total;
683     threadParams_t *threadParams = (threadParams_t *)threadp;
684
685     read_time(&start);
686     syslog(LOG_CRIT, "Task 5, Processed Image Save to File thread @ msec=%f \n",
start);
687     printf("Task 5, Processed Image Save to File thread @ msec=%f \n", start);
688
689 }
```

```
690     while (!abortS5)
691     {
692         sem_wait(&semS5);
693
694         execution_cycle[4]++;
695         read_time(&start);
696         syslog(LOG_CRIT, "Task 5, Processed Image Save to File start %d @ msec=%f",
execution_cycle[4], start);
697         fibTest(ITERATION_COUNT);
698         read_time(&end);
699         total = end - start;
700         if(total > wcet[4]) wcet[4] = total;
701         execution_time[4] += total;
702         syslog(LOG_CRIT, "Task 5, Processed Image Save to File Execution complete @
msec=%f, execution time : %f ms\n", end, total);
703     }
704
705     pthread_exit((void *)0);
706
707 }
708
709 void *Service_6(void *threadp)
710 {
711     double start, end, total;
712     threadParams_t *threadParams = (threadParams_t *)threadp;
713
714     read_time(&start);
715     syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote thread @ msec=%f \n",
start);
716     printf("Task 6, Send Time-stamped Image to Remote thread @ msec=%f \n", start);
717
718     while (!abortS6)
719     {
720         sem_wait(&semS6);
721
722         execution_cycle[5]++;
723         read_time(&start);
724         syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote start %d @ msec=
%f", execution_cycle[5], start);
725         fibTest(ITERATION_COUNT);
726         read_time(&end);
727         total = end - start;
728         if(total > wcet[5]) wcet[5] = total;
729         execution_time[5] += total;
730         syslog(LOG_CRIT, "Task 6, Send Time-stamped Image to Remote Execution
complete @ msec=%f, execution time : %f ms\n", end, total);
731     }
732
733     pthread_exit((void *)0);
734
735 }
736
737 void *Service_7(void *threadp)
738 {
739     double start, end, total;
```

```

742     threadParams_t *threadParams = (threadParams_t *)threadp;
743
744     read_time(&start);
745     syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug thread @ msec=%f \n", start);
746     printf("Task 7, 10 sec Tick Debug Thread @ msec=%f \n", start);
747
748     while (!abortS7)
749     {
750         sem_wait(&semS7);
751
752         execution_cycle[6]++;
753         read_time(&start);
754         syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug start %d @ msec=%f",
execution_cycle[6], start);
755         fibTest(ITERATION_COUNT);
756         read_time(&end);
757         total = end - start;
758         if(total > wcet[6]) wcet[6] = total;
759         execution_time[6] += total;
760         syslog(LOG_CRIT, "Task 7, 10 sec Tick Debug Execution complete @ msec=%f,
execution time : %f ms\n", end, total);
761     }
762
763     pthread_exit((void *)0);
764 }
765
766 double getTimeMsec(void)
767 {
768     struct timespec event_ts = {0, 0};
769
770     clock_gettime(CLOCK_MONOTONIC, &event_ts);
771     return ((event_ts.tv_sec) * 1000.0) + ((event_ts.tv_nsec) / 1000000.0);
772 }
773
774 void print_scheduler(void)
775 {
776     int schedType;
777
778     schedType = sched_getscheduler(getpid());
779
780     switch (schedType)
781     {
782     case SCHED_FIFO:
783         printf("Pthread Policy is SCHED_FIFO\n");
784         break;
785     case SCHED_OTHER:
786         printf("Pthread Policy is SCHED_OTHER\n");
787         exit(-1);
788         break;
789     case SCHED_RR:
790         printf("Pthread Policy is SCHED_RR\n");
791         exit(-1);
792         break;
793     default:
794         printf("Pthread Policy is UNKNOWN\n");
795         exit(-1);
796     }

```

797 | }
798 |

main.c

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <stdbool.h>
4  #include "main.h"
5  #include "drivers/pinout.h"
6  #include "utils/uartstdio.h"
7
8  // TivaWare includes
9  #include "driverlib/sysctl.h"
10 #include "driverlib/debug.h"
11 #include "driverlib/rom_map.h"
12 #include "driverlib/rom.h"
13 #include "driverlib/timer.h"
14 #include "driverlib/inc/hw_memmap.h"
15 #include "driverlib/inc/hw_ints.h"
16
17 // FreeRTOS includes
18 #include "FreeRTOSConfig.h"
19 #include "FreeRTOS.h"
20 #include <timers.h>
21 #include <semphr.h>
22 #include "task.h"
23 #include "queue.h"
24 #include "limits.h"
25
26 #define FIB_LIMIT_FOR_32_BIT 47
27 #define ITERATION 1200
28 #define MULTIPLIER 10
29 #define Hz (30 * MULTIPLIER) // Hz
30 #define SEQUENCER_COUNT 900
31 #define UART_BAUD_RATE 1000000
32
33 SemaphoreHandle_t task_1_SyncSemaphore, task_2_SyncSemaphore, task_3_SyncSemaphore,
task_4_SyncSemaphore, task_5_SyncSemaphore, task_6_SyncSemaphore,
task_7_SyncSemaphore;
34 TickType_t startTimeTick;
35 TaskHandle_t Task1_handle, Task2_handle, Task3_handle, Task4_handle, Task5_handle,
Task6_handle, Task7_handle;
36 uint32_t counter_isr = 0;
37 uint32_t ulPeriod;
38 volatile bool abort_test = false;
39 uint32_t wcet[7];
40 uint32_t execution_time[7];
41 uint32_t execution_cycle[7];
42
43 void init_Timer();
44 void init_Uart();
45 void init_Clock();
46
47 void fibonacci()
48 {
49     uint32_t i,j;
50     uint32_t fib = 1, fib_a = 1, fib_b = 1;
51     for ( i=0; i<ITERATION; i++)
```

```

52     {
53         for(j=0; j<FIB_LIMIT_FOR_32_BIT; j++){
54             fib_a = fib_b;
55             fib_b = fib;
56             fib = fib_a + fib_b;
57         }
58     }
59 }
60 }
61
62
63 void print_data(){
64     uint32_t i = 0;
65     for (i = 0; i < 7; i++){
66         UARTprintf("***** Task %d wcet %d total_executation_time %d execution unit %d
67         *****\n\r", i+1, wcet[i], execution_time[i], execution_cycle[i]);
68     }
69 }
70 void Timer0Isr_Sequencer(void)
71 {
72     TickType_t xCurrentTick = xTaskGetTickCount();
73     ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // Clear the timer interrupt
74     counter_isr++;
75
76     UARTprintf("Sequencer Thread ran at %d ms and Cycle of sequencer %d \n\r",
77     xCurrentTick, counter_isr);
78
79     if ((counter_isr % 10) == 0)
80     {
81         // Service_1 = RT_MAX-1 @ 300 Hz
82         xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
83     }
84
85     if ((counter_isr % 30) == 0)
86     {
87         // Service_2 = RT_MAX-2 @ 100 Hz
88         xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis
89         thread
90         // Service_4 = RT_MAX-2 @ 100 Hz
91         xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
92         // Service_6 = RT_MAX-2 @ 100 Hz
93         xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote
94         thread
95     }
96
97     if ((counter_isr % 60) == 0)
98     {
99         // Service_3 = RT_MAX-3 @ 50 Hz
100         xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
101         // Service_5 = RT_MAX-3 @ 50 Hz
102         xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
103     }
104
105     if ((counter_isr % 300) == 0)

```

```

105     {
106         // Service_7 = RT_MIN    10 Hz
107         xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
108     }
109
110     if (counter_isr > SEQUENCER_COUNT)
111     {
112         xSemaphoreGive(task_1_SyncSemaphore); // Frame Sampler thread
113         xSemaphoreGive(task_2_SyncSemaphore); // Time-stamp with Image Analysis
114         thread
115         xSemaphoreGive(task_3_SyncSemaphore); // Difference Image Proc thread
116         xSemaphoreGive(task_4_SyncSemaphore); // Time-stamp Image Save to File thread
117         xSemaphoreGive(task_5_SyncSemaphore); // Processed Image Save to File thread
118         xSemaphoreGive(task_6_SyncSemaphore); // Send Time-stamped Image to Remote
119         thread
120         xSemaphoreGive(task_7_SyncSemaphore); // 10 sec Tick Debug thread
121         abort_test = true;
122         ROM_TimerDisable(TIMER0_BASE, TIMER_A);
123         print_data();
124     }
125 }
126
127 // Process 1
128 void xTask1(void *pvParameters)
129 {
130     BaseType_t xResult;
131
132     while (!abort_test)
133     {
134         xResult = xSemaphoreTake(task_1_SyncSemaphore, portMAX_DELAY);
135
136         if (xResult == pdPASS)
137         {
138             execution_cycle[0]++;
139             TickType_t xCurrentTick = xTaskGetTickCount();
140             UARTprintf("Task 1 (Frame Sampler thread) Start Time:%d ms, Release count\n\r", xCurrentTick, execution_cycle[0]);
141             fibonacci();
142             TickType_t xFibTime = xTaskGetTickCount();
143             TickType_t total_time = (xFibTime - xCurrentTick);
144             execution_time[0] += total_time;
145             if(wcet[0] < total_time) wcet[0] = total_time;
146
147             UARTprintf("Task 1 (Frame Sampler thread) Completion Time:%d ms, Execution Time:%d ms\n\r", xFibTime, total_time);
148         }
149     }
150     vTaskDelete( NULL );
151 }
152
153 void xTask2(void *pvParameters)
154 {
155     BaseType_t xResult;
156
157     while (!abort_test)
158     {

```

```

158     xResult = xSemaphoreTake(task_2_SyncSemaphore, portMAX_DELAY);
159
160
161     if (xResult == pdPASS)
162     {
163         execution_cycle[1]++;
164         TickType_t xCurrentTick = xTaskGetTickCount();
165         ms, Release count %d \n\r", xCurrentTick, execution_cycle[1]);
166         fibonacci();
167         TickType_t xFibTime = xTaskGetTickCount();
168         TickType_t total_time = (xFibTime - xCurrentTick);
169         execution_time[1] += total_time;
170         if(wcet[1] < total_time) wcet[1] = total_time;
171         Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
172     }
173 }
174 vTaskSuspend( NULL );
175 }
176 void xTask3(void *pvParameters)
177 {
178     BaseType_t xResult;;
179
180     while (!abort_test)
181     {
182
183         xResult = xSemaphoreTake(task_3_SyncSemaphore, portMAX_DELAY);
184
185         if (xResult == pdPASS)
186         {
187             execution_cycle[2]++;
188             TickType_t xCurrentTick = xTaskGetTickCount();
189             Release count %d \n\r", xCurrentTick, execution_cycle[2]);
190             fibonacci();
191             TickType_t xFibTime = xTaskGetTickCount();
192             TickType_t total_time = (xFibTime - xCurrentTick);
193             execution_time[2] += total_time;
194             if(wcet[2] < total_time) wcet[2] = total_time;
195             Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
196         }
197     }
198     vTaskDelete( NULL );
199 }
200 void xTask4(void *pvParameters)
201 {
202     BaseType_t xResult;
203
204     while (!abort_test)
205     {
206
207         xResult = xSemaphoreTake(task_4_SyncSemaphore, portMAX_DELAY);
208
209         if (xResult == pdPASS)
210         {

```

```

211         execution_cycle[3]++;
212         TickType_t xCurrentTick = xTaskGetTickCount();
213         ms, Release count %d \n\r", xCurrentTick, execution_cycle[3]);
214         fibonacci();
215         TickType_t xFibTime = xTaskGetTickCount();
216         TickType_t total_time = (xFibTime - xCurrentTick);
217         execution_time[3] += total_time;
218         if(wcet[3] < total_time) wcet[3] = total_time;
219         Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
220     }
221 }
222 vTaskDelete( NULL );
223 }
224 void xTask5(void *pvParameters)
225 {
226     BaseType_t xResult;
227
228     while (!abort_test)
229     {
230
231         xResult = xSemaphoreTake(task_5_SyncSemaphore, portMAX_DELAY);
232
233         if (xResult == pdPASS)
234         {
235             execution_cycle[4]++;
236             TickType_t xCurrentTick = xTaskGetTickCount();
237             ms, Release count %d \n\r", xCurrentTick, execution_cycle[4]);
238             fibonacci();
239             TickType_t xFibTime = xTaskGetTickCount();
240             TickType_t total_time = (xFibTime - xCurrentTick);
241             execution_time[4] += total_time;
242             if(wcet[4] < total_time) wcet[4] = total_time;
243             Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
244         }
245     }
246     vTaskDelete( NULL );
247 }
248 void xTask6(void *pvParameters)
249 {
250     BaseType_t xResult;
251
252     while (!abort_test)
253     {
254
255         xResult = xSemaphoreTake(task_6_SyncSemaphore, portMAX_DELAY);
256
257         if (xResult == pdPASS)
258         {
259             execution_cycle[5]++;
260             TickType_t xCurrentTick = xTaskGetTickCount();
261             Time:%d ms, Release count %d \n\r", xCurrentTick, execution_cycle[5]);
262             fibonacci();

```

```

263     TickType_t xFibTime = xTaskGetTickCount();
264     TickType_t total_time = (xFibTime - xCurrentTick);
265     execution_time[5] += total_time;
266     if(wcet[5] < total_time) wcet[5] = total_time;
267     UARTprintf("Task 6 (Send Time-stamped Image to Remote thread) Completion
Time:%d ms, Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
268 }
269 }
270 vTaskDelete( NULL );
271 }
272 void xTask7(void *pvParameters)
273 {
274     BaseType_t xResult;
275
276     while (!abort_test)
277     {
278
279         xResult = xSemaphoreTake(task_7_SyncSemaphore, portMAX_DELAY);
280
281         if (xResult == pdPASS)
282         {
283             execution_cycle[6]++;
284             TickType_t xCurrentTick = xTaskGetTickCount();
285             UARTprintf("Task 7 (10 sec Tick Debug thread) Start Time:%d ms , Release
count %d \n\r", xCurrentTick, execution_cycle[6]);
286             fibonacci();
287             TickType_t xFibTime = xTaskGetTickCount();
288             TickType_t total_time = (xFibTime - xCurrentTick);
289             execution_time[6] += total_time;
290             if(wcet[6] < total_time) wcet[6] = total_time;
291             UARTprintf("Task 7 (10 sec Tick Debug thread) Completion Time:%d ms,
Execution Time:%d ms\n\r", xFibTime, (xFibTime - xCurrentTick));
292         }
293     }
294     vTaskDelete( NULL );
295 }
296
297 // Main function
298 int main(void)
299 {
300     init_Clock();
301     init_Uart();
302     init_Timer();
303
304     task_1_SyncSemaphore = xSemaphoreCreateBinary();
305     task_2_SyncSemaphore = xSemaphoreCreateBinary();
306     task_3_SyncSemaphore = xSemaphoreCreateBinary();
307     task_4_SyncSemaphore = xSemaphoreCreateBinary();
308     task_5_SyncSemaphore = xSemaphoreCreateBinary();
309     task_6_SyncSemaphore = xSemaphoreCreateBinary();
310     task_7_SyncSemaphore = xSemaphoreCreateBinary();
311
312     UARTprintf("Cyclic executer : %d Hz\n\r", Hz);
313     xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 4, &Task1_handle);
314     xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 3, &Task2_handle);
315     xTaskCreate(xTask3, "Task3", configMINIMAL_STACK_SIZE, NULL, 2, &Task3_handle);
316     xTaskCreate(xTask4, "Task4", configMINIMAL_STACK_SIZE, NULL, 3, &Task4_handle);

```

```

317     xTaskCreate(xTask5, "Task5", configMINIMAL_STACK_SIZE, NULL, 2, &Task5_handle);
318     xTaskCreate(xTask6, "Task6", configMINIMAL_STACK_SIZE, NULL, 3, &Task6_handle);
319     xTaskCreate(xTask7, "Task7", configMINIMAL_STACK_SIZE, NULL, 1, &Task7_handle);
320
321     startTimeTick = xTaskGetTickCount();
322
323     vTaskStartScheduler();
324     UARTprintf("\nTEST COMPLETE\n");
325     return (0);
326 }
327
328 void init_Timer()
329 {
330     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
331     ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);           // 32 bits Timer
332     TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr_Sequencer); // Registering isr
333
334     ulPeriod = (SYSTEM_CLOCK / Hz);
335     ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod - 1);
336
337     ROM_TimerEnable(TIMER0_BASE, TIMER_A);
338     ROM_IntEnable(INT_TIMER0A);
339     ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
340 }
341
342 void init_Clock()
343 {
344     // Initialize system clock to 120 MHz
345     uint32_t output_clock_rate_hz;
346     output_clock_rate_hz = ROM_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
347     SYSCTL_OSC_MAIN | SYSCTL_USE_PCL | SYSCTL_CFG_VCO_480), SYSTEM_CLOCK);
348     ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
349 }
350
351 void init_Uart()
352 {
353     // Initialize the GPIO pins for the Launchpad
354     PinoutSet(false, false);
355     UARTStdioConfig(0, UART_BAUD_RATE, SYSTEM_CLOCK);
356 }
357
358 /* ASSERT() Error function
359 *
360 * failed ASSERTS() from driverlib/debug.h are executed in this function
361 */
362 void __error__(char *pcFilename, uint32_t ui32Line)
363 {
364     // Place a breakpoint here to capture errors until logging routine is finished
365     while (1)
366     {
367     }
368 }

```

