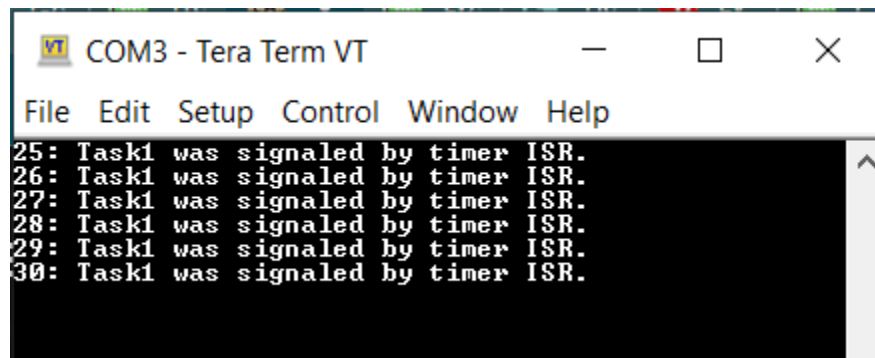


Q1

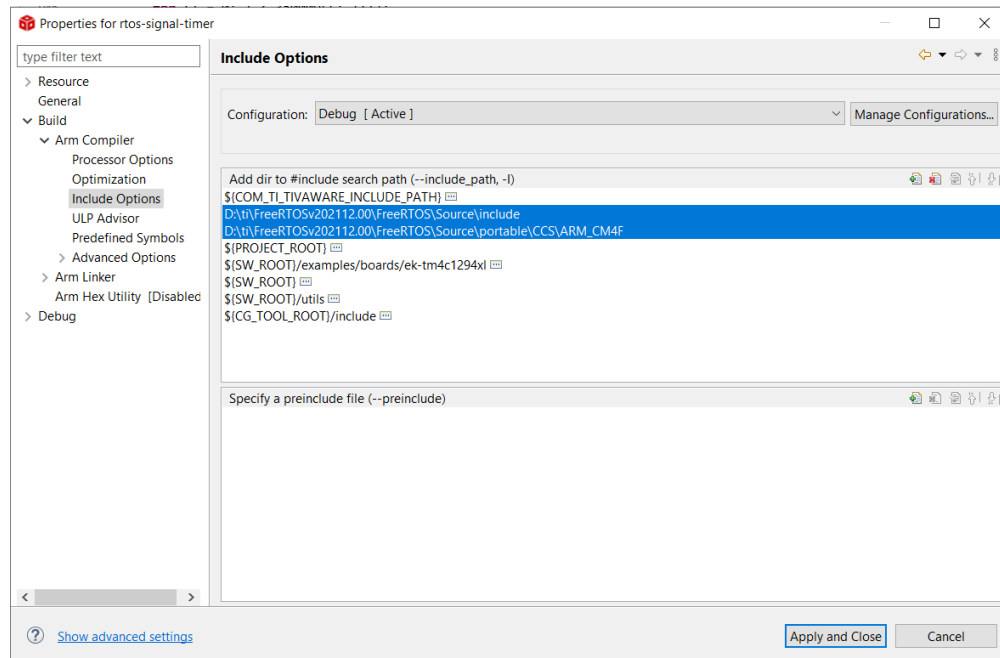
For this question, I learned about the EK-TM4C1294XL development kit from Texas Instruments. I first had to install Code Composer Studio (CCS) and the TivaWare C Series in order to leverage the library/drivers for the timer peripheral and UART print functions. To start, I figured out how to use timer0 and set up interrupts so that I could signal a simple function as to when it could execute. The timer ISR was fired every 1 second. The task (named Task1) prints to the console every time it is able to execute. The numbers on the far left indicate the global time (in seconds). Global time is being generated by the timer0 ISR by incrementing a variable.



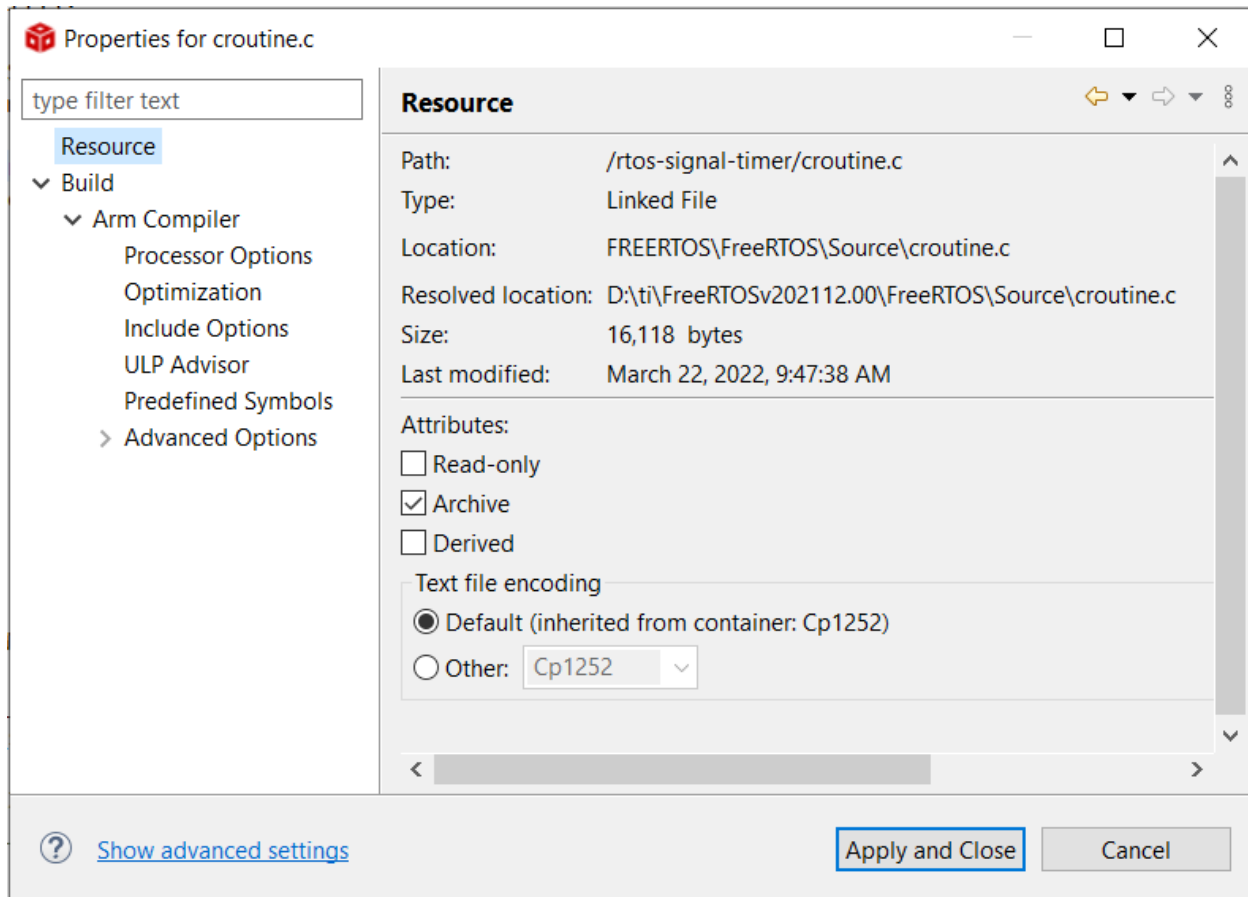
```
COM3 - Tera Term VT
File Edit Setup Control Window Help
25: Task1 was signaled by timer ISR.
26: Task1 was signaled by timer ISR.
27: Task1 was signaled by timer ISR.
28: Task1 was signaled by timer ISR.
29: Task1 was signaled by timer ISR.
30: Task1 was signaled by timer ISR.
```

Q2

For this problem, I first had to install FreeRTOS. I attempted to follow the instructions provided but ran into many issues while attempting to build. In the end, I downloaded FreeRTOSv202112.00 from the FreeRTOS website. I unpacked this source code in the ti installation directory. I started a new blinky project and tried including basic FreeRTOS headers. I ran into compile errors and tackled these one by one. The first major step to overcome was adding the 'include' paths for the compiler to use. The paths which were added for my machine are highlighted in the screenshot below.



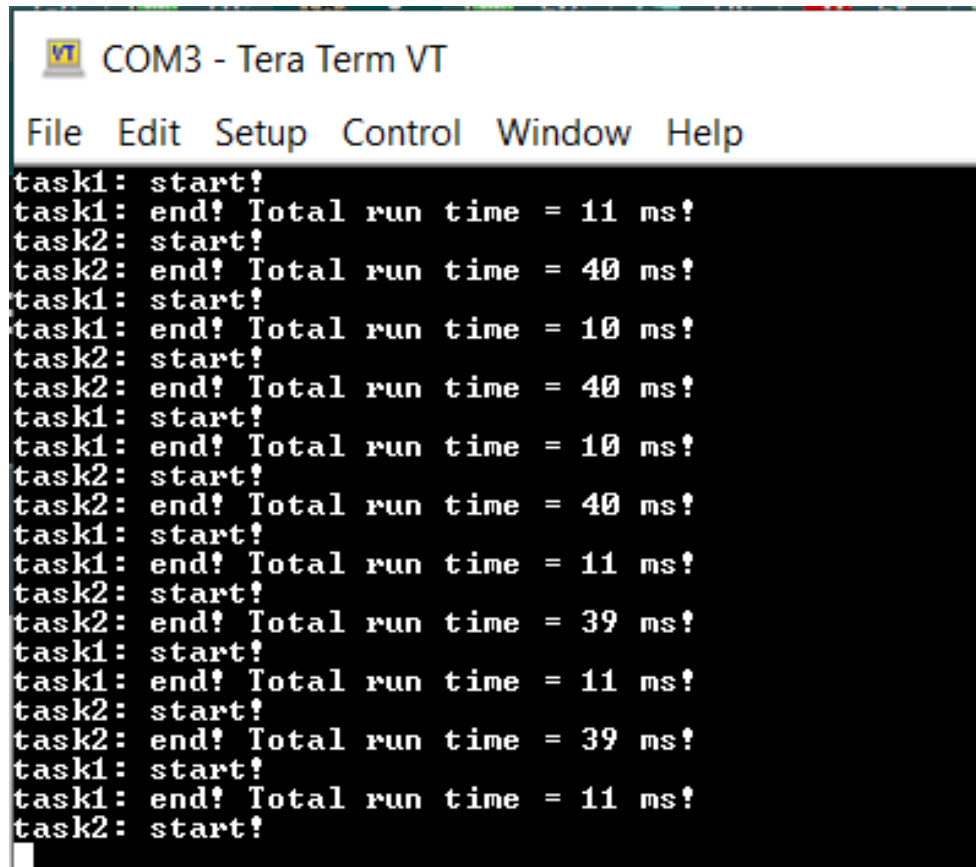
These paths point to the include directory for FreeRTOS and to the 'portable' directory to be used. The next challenge was adding the source files for FreeRTOS, this was accomplished by right clicking on the project, and selecting the 'Add Files' option. This prompts the user to then select the desired files, and either copy or link to the project. I opted to link to the project. An example of this is shown bellow for croutine.c.



Notice that FREERTOS variable was declared which expands to 'D:\ti\FreeRTOSv202112.00'. This is specific to my machine, so someone wishing to use this code on their machine will need to redefine FREERTOS on their machine.

In general, this was the procedure used to get FreeRTOS working on my machine. I also might have added a few more files using the method discussed above for portasm.asm and heap_1.c. The other thing I need to mention is the creation of FreeRTOSConfig.h. This file was copied from the sample freeRTOS project provided in the zip on Canvas. I simply copied it to the project.

Once this was all complete and FreeRTOS was compiling, I created two tasks (task1 and task2) and two binary semaphores (t1_sem and t2_sem). Prior to starting the taskScheduler, I give the t1_sem so that task1 is able to run. Task2 is unable to run until task1 gives the t2_sem. Task1 gives the t2_sem when it completes. Task1 is subsequently unable to run until task2 gives t1_sem at the end of its processing. This back and forth of semaphores repeats infinitely. In the screenshot below, we can see that task1 runs for ~10ms and task2 runs for ~40ms. Neither task interrupts each other as a result of the binary semaphore 'hand off' procedure just described.



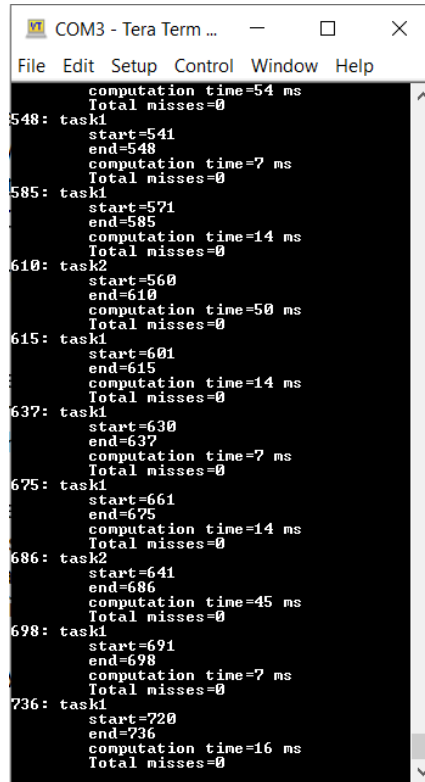
```
COM3 - Tera Term VT
File Edit Setup Control Window Help
task1: start!
task1: end! Total run time = 11 ms!
task2: start!
task2: end! Total run time = 40 ms!
task1: start!
task1: end! Total run time = 10 ms!
task2: start!
task2: end! Total run time = 40 ms!
task1: start!
task1: end! Total run time = 10 ms!
task2: start!
task2: end! Total run time = 40 ms!
task1: start!
task1: end! Total run time = 11 ms!
task2: start!
task2: end! Total run time = 39 ms!
task1: start!
task1: end! Total run time = 11 ms!
task2: start!
task2: end! Total run time = 39 ms!
task1: start!
task1: end! Total run time = 11 ms!
task2: start!
```

Q3

For question 3, we combined the concept of the timer ISR, and FreeRTOS tasks. In this case, the goal was to have the timer ISR signal to task1 and task2 when they could run at a specified frequency. Task1 can run every 30ms and task2 is able to run every 80ms. Each task maintains its workload from Q2 (10ms and 40ms respectively). It is the job of the timer ISR to signal to each task when it can run at a specific frequency. To accomplish this, we first set the timer0 ISR frequency to be every 1ms. Then we obtain the ticks since the scheduler was started. This tick count is converted to milliseconds and used to determine if task1 or task2 can be run. If a task is ready to be run again, the timer ISR will give the respective binary semaphore for the task. If it is time for a task to run, but the task is still running, we consider that a deadline miss, and increment a global variable which keeps count of the total misses for that task.

At the end of each tasks computation, it prints out details about it's most recent run. Information such as the start, end, computation time, and total number of misses at that point, is printed to the console. We ran into the issue of the tasks interrupting each other during printing, so a third semaphore was introduced to manage access to printing to the console. This added some additional overhead and so

the tasks busy loads/computation tasks were decreased to account. Bellow is a screenshot of the tasks running and printing their stats.



```
COM3 - Tera Term ...
File Edit Setup Control Window Help

computation time=54 ms
Total misses=0
548: task1
    start=541
    end=548
    computation time=7 ms
    Total misses=0
585: task1
    start=571
    end=585
    computation time=14 ms
    Total misses=0
610: task2
    start=560
    end=610
    computation time=50 ms
    Total misses=0
615: task1
    start=601
    end=615
    computation time=14 ms
    Total misses=0
637: task1
    start=630
    end=637
    computation time=7 ms
    Total misses=0
675: task1
    start=661
    end=675
    computation time=14 ms
    Total misses=0
686: task2
    start=641
    end=686
    computation time=45 ms
    Total misses=0
698: task1
    start=691
    end=698
    computation time=7 ms
    Total misses=0
736: task1
    start=720
    end=736
    computation time=16 ms
    Total misses=0
```

From here we observe that task1 started at time 541ms (relative to the scheduler start time), and ended at time 548 ms. Thus the total computation time was ~7ms. The next time task1 will run is at time 571 ms, which is approximately 30ms later. This shows that the timer0 ISR is properly dispatching task1 every 30ms. This also shows that task1 is computing for approximately 10ms.

Let's do the same for task2. The first sample of task2 in the screenshot is at time 560 ms. It ends at 610 ms, for a total of 50ms computation time. This is a 10 seconds more than the desired 40ms computation time, but still within the 80ms deadline. We see that the next time task2 runs is at 641 ms, which is approximately 80ms later. This again shows that the timer0 ISR is correctly dispatching the tasks at the desired frequency.