This exam is open book, open notes. Calculators are allowed. Internet access is not permitted. Show all work for full credit. Partial credit is also assigned.

1) [12 pts] Answer the following questions about real-time debugging.
   a) [6 pts] One of your Tasks is triggered by a periodic **hardware timer** (through an ISR). How would you measure the response time of this Task?

- **Determine the time elapsed from the interrupt to the end of the event handler using the current timer count. This is the instantaneous response time.**
- **Keep track of the max response time in a global. Update it if the instantaneous response time is higher. Exercise code paths that may affect this task's response time.**

Attempts to calculate the response time from RTOS benchmarks and other components received penalties because there was not enough information given with the problem description to prepare such a calculation. Particularly, the priority of the task was not mentioned. Therefore we cannot assume that the response time is simply the highest priority task latency plus execution time (even if ignoring preemptions by ISRs). Even the highest priority task latency depends on the number of ISRs and the length of some critical sections.

   b) [6 pts] How would you measure the **worst case** execution time of a C function using a logic analyzer? Assume the execution time of this function varies from run to run.

- **Set a GPIO output to 1 before calling this function.**
- **Reset the GPIO output to 0 after this function returns.**
- **Run this code many times in a loop. Each iteration should have a delay until the next one starts to avoid ambiguity of edges.**
- **Trigger the logic analyzer on the rising edge of the GPIO signal.**
- **Turn on infinite persist mode and find the time from the trigger to the farthest falling edge.**

2) [30 pts] Trace the execution of the following multitasking code on a single-CPU system until all tasks block, requiring action beyond this setup to unblock. The tasks start execution at the points labeled in the comments. A **single Hwi0 interrupt** occurs during the first execution of `<task2 code>`, as indicated in the comments. Fill in the table on the next page (you may leave table cells blank if unchanged from the previous row). The initial task and semaphore state is in the first row of the table. The semaphores are counting, and unblock tasks in FIFO order. Semaphore actions, scheduler actions and each execution of `<task1 code>` and `<task2 code>` should be separate steps. You do not need to label the steps in the code.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
    <clear interrupt flag>;
    Semaphore_post(sem1);
    Semaphore_post(sem1); // posting to the same semaphore twice
}

void task1(UArg arg0, UArg arg1) {  // high priority
    // <- start
    Semaphore_post(sem2);
    while (1) {
        Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
        <task1 code>;
    }
}

void task2(UArg arg0, UArg arg1) {  // low priority
    // <- start
    while (1) {
        Semaphore_pend(sem2, BIOS_WAIT_FOREVER);
        <task2 code>; // <-Hwi0 interrupts during first loop iteration
    }
}
```

Step	Action	task1	task2	sem1		sem2	
				count	wait list	count	wait list
0	Start	Ready	Ready	0	-	0	-
1	**scheduler runs**	**Running**	**Ready**				
2	**task1 posts to sem2**					**1**	
3	**task1 pends on sem1**	**Blocked**			**task1**		
4	**scheduler runs**		**Running**				
5	**task2 pends on sem2**					**0**	
6	**<task2 code> starts running**						
7	**Hwi0 interrupts, posts to sem1**	**Ready**			**-**		
8	**Hwi0 posts to sem1 again**			**1**			
9	**scheduler runs when Hwi0 exits**	**Running**	**Ready**				
10	**<task1 code> runs**						
11	**task1 pends on sem1**			**0**			
12	**<task1 code> runs**						
13	**task1 pends on sem1**	**Blocked**			**task1**		
14	**scheduler runs**		**Running**				
15	**<task2 code> finishes**						
16	**task2 pends on sem2**		**Blocked**				**task2**

3) [30 pts] Two TI-RTOS Tasks (and only these Tasks) share data through the global array `d[]`. You are using a GateTask object to protect the critical sections accessing `d[]`. The semaphores `sem1` and `sem2` are signaled by ISRs. Assume other Tasks of intermediate and higher priority exist in the system.

```
int32_t d[2]; // shared between task1 and task2 only

void task1(UArg arg0, UArg arg1) {  // high priority, short period
    int32_t x[2];
    while (1) {
        Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
        // other code working with x[]

        d[0] = x[0]; // critical section
        d[1] = x[1];
    }
}
void task2(UArg arg0, UArg arg1) {  // low priority
    IArg key;
    int32_t y[2];
    while (1) {
        Semaphore_pend(sem2, BIOS_WAIT_FOREVER);

        key = GateHwi_enter(gate1);
        y[0] = d[0]; // critical section
        y[1] = d[1];
        GateHwi_leave(gate1, key);

        // other code working with y[]
    }
}
```

a) [6 pts] You have neglected to protect the critical section in the high-priority task1. Does this cause any shared data issues? Explain.

No. GateHwi disables Task, Swi and Hwi scheduling. Therefore the critical section in task2 cannot be preempted by any task (including task1). The critical section in task1 cannot be preempted by task2 because task1 is higher priority. Because neither critical section can be preempted by another task using the shared array `d[]`, there would be no shared data issues.

(problem continues on the next page)

b) [12 pts] For each TI-RTOS object in the following table briefly specify the pros and cons (e.g. CPU load, latency effects) **compared to GateHwi** for protecting the **given critical sections**. Assume correct application of each object.

Object	Pros	Cons
Semaphore	**Targeted: does not affect response time of other Tasks, Swi or Hwi that are not using the shared data**	**Higher CPU load** **Priority inversion**
GateMutexPri	**Targeted (same as Semaphore)**	**Higher CPU load**
GateTask	**Does not affect the latency of Hwi or Swi, but affects Task latency**	**none**

c) [6 pts] Which gate object from part (b) is the most appropriate to protect these critical sections? Or is GateHwi the most appropriate? Explain.

GateTask is the most appropriate because the critical section is relatively short, so task latency impact is minimal, and priority inversion is a concern due to existence of other tasks of potentially intermediate priorities. This gate uses less CPU time (so lower CPU load) than GateMutexPri, which may be a concern with the short period task1.

GateMutexPri is also acceptable (with an explanation), but imparts higher CPU load than GateTask. GateHwi is not optimal because it affects Hwi and Swi latency unnecessarily.

d) [6 pts] If you went with GateMutexPri and simply replaced the GateHwi calls with the equivalent GateMutexPri calls, would this cause any shared data issues? Explain.

Yes. GateMutexPri does not prevent the critical section in taks2 from being preempted by task1. task1 can then modify the shared array. When task2 resumes, it is now dealing with the second part of `d[]` inconsistent with the first part (processed before being preempted). This is a shared data bug. To avoid it, the critical section in task1 should be protected by the same gate.

4) [28 pts] Answer the following questions about RTOS features.
   a) [16 pts] You are trying to insert an accurate 5 ms delay into a Task. What is the main advantage and the main disadvantage of calling the TI-RTOS Task_sleep() function versus polling a hardware timer for this purpose?
      - [4 pts] Advantage of Task_sleep()

Advantage: Does not waste CPU time: task is blocked, so other tasks can run.
Disadvantage: Inaccurate

      - [4 pts] Advantage of polling a hardware timer

Advantage: Higher timing accuracy is possible (especially in a high-priority task)
Disadvantage: Burns up CPU time

      - [8 pts] Give a method that combines both of the above advantages. List all the additional TI-RTOS objects (including threads) you will need.

- **Set up a hardware timer for one-shot mode with interrupt on timeout**
- **In the Hwi (ISR) for this timer, post to a Semaphore**
- **In the Task, start the hardware timer, then pend on the Semaphore**

Additional TI-RTOS objects: Hwi, Semaphore

   b) [12 pts] From the following list, select **two** semaphore-related issues/bugs that you think are the most difficult to debug. Explain why.

☑ Forgetting to take a semaphore
☐ Forgetting to release a semaphore
☐ Holding a semaphore too long
☑ Deadlock
☑ Priority inversion

These issues result in intermittent bugs that require an interrupt to occur in a narrow time window. The resulting bugs are difficult to detect in testing. Forgetting to take a semaphore is also difficult to detect because the resulting shared data bugs often have subtle effects. Deadlock, while obvious if it occurs, has a very narrow time window for the interrupt. If critical sections are short, priority inversion may only rarely result in missed deadlines.

Forgetting to release a semaphore typically results in a task not unblocking, which is detectable.

Holding a semaphore too long results in missed deadlines in another task that pends on this semaphore. This is usually detectable, but may be rare. If you argued that code inspection to find shared data bugs is easier than real-time measurements, you received full credit.