ECEn 425 Fall 2015
Midterm 1 Solution

1. (30 pts) Mark each of the following true or false.
a. EEROM is slow to read and write and can be written a limited number of times.

True. See page 37.

b. If ISRs can interrupt tasks, then the operating system is necessarily using *preemptive scheduling*.

False. Preemptive scheduling means that the highest priority ready *task* is next to execute, even if the current task has not voluntarily given up the CPU. This form of scheduling is handled by the RTOS. In contrast, the scheduling of ISRs is handled by the hardware, and that mechanism always gives interrupt code priority over regular (non-interrupt) code – provided, of course, that interrupts are enabled.

c. The non-maskable interrupt is commonly used for the most frequently occurring events in a real-time system.

False. As page 89 explains, the non-maskable interrupt (NMI) is usually used for events well beyond the scope of normal processing. Why is this the case? Think about this: the ISR associated with this interrupt runs even if interrupts are disabled, and that is a problem for a popular technique for dealing with shared data. In consequence, you can't use the NMI for any ISR that shares data with any task code, and that is a significant restriction.

d. Correctly written task code is always a C function that will eventually terminate and return.

False. Correctly written RTOS tasks are never called, they generally do not terminate (unless they delete themselves using a kernel function) and they never return.

e. The operation of reading a global variable is always atomic.

False. If the variable size exceeds the word size of the machine, it may take multiple accesses just to read a variable. See page 100-101 and the class slides.

f. In our class labs, each ISR must explicitly disable interrupts at lower priority levels.

False. Look at your code for lab3 and lab4. We never have to explicitly modify the interrupt mask register in this manner. It is one of the nice freebies that the PIC does for us. After calling the handler, each ISR does disable interrupts, but this affects all interrupts and not just those at lower priority levels.

g. C's **volatile** keyword warns the compiler that a variable may be changed by something the compiler can't see.

True. See page 103.

h. According to the YAK specifications, the dispatcher must always save the context of the current task.

False. This is not part of the YAK specs for the dispatcher, and context-switch scenarios were discussed in class where it would not save the current task's context.

i. Variables declared as **static** within a function are not stored on that task's stack.

True. See page 151 and the class slides.

j. In most flash memories, data can only be written in blocks that are much larger than a word or byte.

True. See page 37.

k. The bp register normally points to a memory location containing the base address of the previous stack frame.

True. See the class slides on stack frame layouts and your own code.

l. A critical section is any code sequence that must be atomic for the system to work properly.

True. See page 98 and the class slides.

m. Watchdog timers are restarted under software control.

True. Software resets or restarts the timer, which should never expire in normal operation. If it does expire, the software is assumed to have crashed, and the timer resets the processor, which allows execution to start over.

n. If a function accesses no global variables and uses no hardware in a non-atomic way, it must be reentrant.

False. To be reentrant, it must also avoid calling functions that are themselves non-reentrant, and we don't know that about this particular function. See pages 148-150 in the text.

o. If a task holds a semaphore too long, deadlock may result.

False. This would be a case of bad programming, but not deadlock – it could be the only semaphore in the system, for example. Deadlock requires circular dependence. See page 166.

2. (6 pts) Define the following terms:
a. *priority inversion*

This problem is discussed on pages 164-165 in the text and illustrated in the class slides. The gist of the idea is that a lower-priority task holds a semaphore (or other resource) that a higher priority task blocks on, but a third task keeps the lower-priority task from running and releasing the semaphore. Why the name? In this scenario, once the higher-priority task blocks on the resource, the true priority or importance of the task holding it is actually higher than the statically assigned priority would indicate. Of course, even without bumping up the priority of the task holding the resource, most of the time it will be able to run, release the semaphore, and cause the higher-priority task to unblock. (This assumes that we follow good programming practice and don't hold semaphores for too long.) Note that it is not an indication that something is broken if a higher-priority task is waiting for a lower-priority task to release a semaphore – this would be a fairly common occurrence. The problem arises because the holding task can't run and release it.

b. *semaphore*

In an RTOS, a semaphore is a globally visible shared variable accessed only indirectly through kernel functions (creating, obtaining, and releasing it) that can be used to enforce mutual exclusion or to provide a signal for synchronization. If the function is called to obtain a semaphore and it is not

available, the caller will be blocked until the semaphore becomes available (when it is released by the task holding it).

3. (10 pts) From a log file of a system running the YAK kernel, you notice that, at a particular point in time, control switches from task A to task B. Circle each action below that could have caused the switch to task B.

B was made ready by the tick handler.	A pended on a semaphore.
B forced A to block.	A posted to a semaphore.
B moved itself to the ready list.	A delayed itself.
An interrupt handler forced A to block.	A created a new task.
An interrupt handler posted to a semaphore.	An interrupt handler pended on a semaphore.

Let's consider the possibilities in order. 1. We don't know anything about the relative priorities of A and B. B may have been made ready by the tick handler, and then it would preempt A if B had higher priority, so the first listed action could have caused the switch. 2. There is simply no way that B can force A to block – tasks can't do this, and B wasn't running anyway. 3. Similarly, B can't change its own status to ready, as this is an operation performed by the RTOS, and B wasn't running anyway. 4. If an interrupt handler posted to a semaphore, that could cause B to unblock and preempt A if B has higher priority. 5. Task A could have pended on a semaphore that was not available, causing it to block and allowing B to run. 6. If A posted to a semaphore that B was blocked on, then the post would have caused B to unblock and B would preempt A if B had higher priority. 7. Task A could have delayed itself, allowing B to run. 8. Finally, it is forbidden to have interrupt handlers pend on semaphores (interrupt code can't be blocked like task code), so this is not something you'd observe in an operational system with any BYU grads involved.

4. (6 pts) What solutions exist for the shared-data problem when the sharing is between task and ISR code?

Of the primary alternatives we discussed (disabling interrupts, using semaphores, locking the scheduler), it should be easy to see that disabling interrupts is the only alternative when sharing is between task and interrupt code. (Interrupt code cannot take a semaphore, as it cannot be blocked as task code is blocked. Locking the scheduler has no impact on whether interrupt code executes or not.) One should not, however, neglect the other alternatives discussed in the text beginning on page 107. These include double buffering, the use of a circular queue, and repeatedly reading the value until it is the same twice. They can be thought of as "programming tricks" that certainly could be made to work, but they probably wouldn't be your first choices without compelling reasons to avoid disabling interrupts.

5. (5 pts) From the programmer's point of view, what are the practical implications of a function being *reentrant* or not?

Note that I wasn't asking for the conditions that you have to test for to see if a function is reentrant (although you got a reasonable chunk of points just for stating this). I was asking why this is something you would care about as you develop code for any concurrent system. The bottom line is that you can call a reentrant function from any task and (assuming it is written correctly, of course) it will always work correctly, regardless of where interrupts (and subsequent switches to other tasks) occur. This makes life much simpler for you. If you need to call a function that is non-reentrant from multiple tasks, you'd have to create some kind of mutual exclusion mechanism at the points of call to make sure only one routine was in that function at a time. This wouldn't be much fun. Note that reentrant functions may not necessarily be called by ISRs for other reasons: they might cause the caller to block (e.g., YKDelayTask(), YKSemPend()).

6. (10 pts) Task A is executing and an interrupt occurs, causing an ISR and handler to run, after which task B runs. From the last instruction of A to execute (before the interrupt) to the first instruction of B to execute (after the ISR completes), specify the order in which the actions below occur. (Write "1" next to the first to occur, "2" for the second, etc.) If a specific action would not necessarily occur in this scenario, write an "X" next to it. If an event occurs multiple times, consider its first occurrence only. Assume the system is running YAK and that it supports nested interrupts.

a. Enable interrupts	f. Execute cli instruction
b. Execute EOI command	g. Call handler
c. Call YKExitISR()	h. Call YKEnterISR()
d. Save flag register	i. Save register AX
e. Restore register AX	j. Execute iret instruction

The sequence starts with the hardware responding to the interrupt. This disables interrupts (not a choice listed), saves some state including the flag (d), and starts executing instructions in the appropriate ISR. The software then saves remaining context (i) and calls the ISR entry routine (h) before enabling interrupts (a) to support interrupt nesting. At this point it calls the handler (g), which responds to the event associated with the interrupt.

The remaining steps are essentially the ISR finishing up and passing control back to a task. It first disables interrupts (f) for this critical housekeeping, executes the EOI command (b) so the PIC can mark the end of processing this level of interrupt code, and calls the ISR exit routine (c). (Step b must occur before step c because control will not return to the ISR code past the point of calling YKExitISR() if preemption requires another task to execute.) In this case, the call to the scheduler transfers control to a different task, so that task's context will be restored (including restoring its version of all registers) (e) and control will then be passed to that task by executing an iret (j).

7. (5 pts) RTOS software can change task priorities dynamically. Could it also dynamically change the priority of an interrupt associated with a particular input device? Explain how it could do this, or why it cannot be done.

In conventional microcontrollers, interrupt priority levels are determined by which pin on the microprocessor (or PIC in the case of the 8086) the device is connected to. If you want to change the priority level of a particular device, you have to rewire it and change its physical connection. Note that changing the interrupt vector table simply changes the software routine (ISR) that runs when an interrupt at a particular priority level is responded to – it does not change the priority level associated with interrupts caused by a particular device, so long as it is connected to the same input pin on the microprocessor.

8. (8 pts) Assuming normal YAK functionality, when the dispatcher runs for the first time, which of these kernel functions must already have been called at least once? (Circle each correct answer.)

YKNewTask()	YKRun()
YKScheduler()	YKEnterISR()
YKInitialize()	YKSemCreate()
YKDelayTask()	YKTickHandler()

By "normal YAK functionality" I mean according to the YAK specifications on the web page and the examples of YAK code used in class. The dispatcher runs for the first time within YKRun() at the end of main(), since this call causes task code to run for the first time. At that point (assuming correctly written code), main() will have called YKInitialize() and YKRun(). The code in main() will also have created at least one task, so YKNewTask() will have been called at least twice at that point, including the call to create the idle task. (The YAK specs state: "At least one user-defined task

must be created with a call to YKNewTask before YKRun is called. “) Similarly, YKRun calls YKScheduler() which calls the dispatcher, so YKScheduler() will have been called exactly once. (From the YAK webpage: “This routine [YKRun()] causes the scheduler to run for the first time (which then calls the dispatcher).”)

Until YKRun() is called, the system is not really ready to start normal operations, and you probably don’t want to start dealing with interrupts, so it would be a bad thing if YKExitISR() or YKTickHandler() had already run once. YKSemCreate() should have been called if the application requires a semaphore (best to do initialization of semaphores in main()), but not all applications require semaphores. Finally, YKDelayTask() will not have been called yet since it is called only by task code and no task has yet executed.

9. (10 pts) Circle each YAK function below that must include a call to the scheduler in its source code, according to the kernel specifications.

YKNewTask()	YKRun()
YKSemPost()	YKEnterISR()
YKInitialize()	YKSemCreate()
YKDelayTask()	YKTickHandler()
YKExitISR()	YKSemPend()

In general, we need a call to the scheduler at the end of any kernel function that might have changed the status of any task in the system. (The call to the scheduler could still be conditional, based on whether it is called from interrupt code or task code, for example.) This certainly includes functions typically called by task code, including YKNewTask(), YKSemPost(), YKDelayTask(), and YKSemPend(). Let’s consider the other cases: YKInitialize() is called only once, at the start of main() in application code. It does not call the scheduler because the system doesn’t start executing task code until YKRun() executes, and that is the last thing done by the code in main(). YKRun() calls the scheduler to get task execution started. YKEnterISR() and YKExitISR() are called in ISR code while interrupts are disabled. We don’t call the scheduler in the former, but we do in the latter (if we’re returning to task code) to make sure we’re executing the highest priority ready task. YKSemCreate() cannot change the status of any task, so it need not call the scheduler. (As discussed in class, it would be a good idea for the application code to create all semaphores in main() before calling YKRun().) Finally, YKTickHandler() is a special case because it can change the status of a task but we do NOT want to call the scheduler at this point. We have to execute the current ISR code to the point of executing the EOI command or our interrupt priority levels will be messed up – the call to the scheduler in YKExitISR() will ensure that we execute the highest priority ready task when we finish the ISR.

10. (5 pts) Can you increase the throughput of a system (e.g., rate at which data is processed) by increasing the frequency of the timer interrupt? Why or why not?

The short answer here is no. There is something close to a fixed overhead associated with the interrupt code for each timer tick, and ramping up the tick frequency simply runs that code more often, resulting in more overhead. If the processor was otherwise busy running task code, this overhead will actually reduce performance. Why isn’t the rate of data processing connected to the timer tick? You could build a system where this happens, but it doesn’t make a lot of sense. The data the system is processing show up via some interface device that almost certainly has an interrupt of its own. To increase throughput, you want to increase the efficiency of responding to those interrupts and processing the associated data.

11. (5 pts) A colleague states: “My professor said ISRs are like hardware-induced function calls, so I know that ISRs and functions have to save the same registers in the same way.” Is this true, or are there important differences in how ISRs and functions save registers? Explain.

The similarity between function calls and ISRs does not extend to the way in which they must save registers. The compiler knows precisely where in the code control is transferred to a function, but an ISR can run between any two given instructions (unless, of course, interrupts are disabled). Therefore, the compiler can manage register usage around function calls, saving caller-saved registers before the call, for example, so they need not be saved within the function. In contrast, the ISR has to save all registers because no assumptions can be made about register status in the interrupted code, and because (in an RTOS) the task may be suspended entirely and another may begin or resume execution.