# ECEn 425 Fall 2018
# Midterm 2 Solution

1. (10 pts) In the box below, write the output generated by this YAK code when it executes. Assume that all semaphore and event references are shown and that all relevant variables and macros are defined correctly elsewhere.

```
#define FLAG1 0x01
#define FLAG2 0x02

void main(void) {
    YKInitialize();
    YKNewTask(Task1,(void *)&Stk1[SSIZE],40);
    YKNewTask(Task2,(void *)&Stk2[SSIZE],37);
    YKNewTask(Task3,(void *)&Stk3[SSIZE],39);
    SemA = YKSemCreate(0);
    Ev1  = YKEventCreate(0);
    YKRun();
}

void Task1(void) {
    printString("A");
    YKEventPend(Ev1,FLAG1,EVENT_WAIT_ANY);
    YKEventReset(Ev1,FLAG1);
    printString("B");
    YKSemPend(SemA);
    printString("C");
    while (1) {
        /* misc. code, no output */
    }
}

void Task2(void) {
    printString("D");
    YKNewTask(Task4,(void *)&Stk4[SSIZE],25);
    printString("E");
    YKSemPend(SemA);
    printString("F");
    while (1) {
        /* misc. code, no output */
    }
}
```

```
void Task3(void) {
    printString("G");
    YKEventPend(Ev1,FLAG1|FLAG2,EVENT_WAIT_ALL);
    YKEventReset(Ev1,FLAG2);
    printString("H");
    YKSemPost(SemA);
    printString("I");
    while (1) {
        /* misc. code, no output */
    }
}

void Task4(void) {
    printString("J");
    YKEventSet(Ev1,FLAG1|FLAG2);
    printString("K");
    YKSemPost(SemA);
    printString("L");
    while (1) {
        /* misc. code, no output */
    }
}
```

PROGRAM OUTPUT:

The program outputs **DJKLEFGHIABC**. This covers a lot of concepts (preemptive scheduling, relative task priorities, semaphore and event conventions, etc.), and most of you nailed it. If you didn't get the right answer, take the time to figure out what part of the system operation you missed. As I mentioned during the exam, it is assumed that all tasks delay themselves or block in the while 1 loops, allowing other tasks to run.

2. (10 pts) Circle each function below that can safely be called by task code AND cause a switch to a different task.

| | | | | |
|---|---|---|---|---|
| YKDelayTask | YKQCreate | YKNewTask | YKEventSet | YKEventPend |
| YKQPend | YKSemPost | YKInitialize | YKSemCreate | YKEventReset |

What should NOT be circled? Calling YKQCreate and YKSemCreate from task code is a bad idea for reasons we've discussed in class, and in any event they cannot cause a context switch. YKInitialize should never be called by a task. Finally YKEventReset can be called by task code, but it does not cause other tasks to unblock (see the YAK specs). You should have circled the rest: YKDelayTask, YKNewTask, YKEventSet, YKEventPend, YKQPend, and YKSemPost.

3. (10 pts) Circle each function below that should NOT be called by interrupt handler code.

| | | | | |
|---|---|---|---|---|
| YKDelayTask | YKQCreate | YKNewTask | YKEventSet | YKEventPend |
| YKQPend | YKSemPost | YKInitialize | YKSemCreate | YKEventReset |

Suppose in your first job you are asked to modify some existing code. The question is this: if you see calls to these functions in interrupt handler code, which would will cause you heartburn? Some will break the kernel, and others are probably "just" a really bad idea. Either way, they should be circled. Let's consider the functions.

YKDelayTask should not be called from interrupt code. Some poor task will get blocked (whatever has the bad luck to be interrupted), and the system will not function as intended. Similarly, interrupt code should never call pend functions, so there should be no calls to YKQPend and YKEventPend. It makes no sense at all to create queues or semaphores in an interrupt routine. Since these routines run in response to external events, how could you guarantee that they were created before any pends or posts by task code? Even if you could guarantee it, why add overhead to an ISR with an action that can't be directly helpful in responding to an event in a timely way? Put the calls to YKQCreate and YKSemCreate in main, or at the very worst in the initial code for a task. By a similar argument, it makes no sense to ever call YKNewTask from a handler. YKInitialize should absolutely NOT be called a second time, and certainly not from interrupt code.

So what *can* you call from interrupt code? It is always okay to post to semaphores and events (pretty typically stuff for ISRs), so YKSemPost and YKEventSet are okay. It can be tricky deciding when events should be reset, and it might take some creativity to come up with a scenario where you might want to reset events in interrupt code (the event reflects the status of a real-world condition that no longer holds?), but I think calls to YKEventReset could occur legitimately in interrupt code.

4. (6 pts) In the context of development tools for embedded systems, what are *segments* and why are they useful?

See the text on page 268. Segments are portions of a program – either code or data – that the locator can treat as a unit and place in memory independent of other pieces. This independent placement is possible because segments are logical groupings of items that all need to be treated in the same manner. They are useful because different parts of the program need to be placed differently: code in ROM, writable data in RAM, main() at the reset address, the interrupt vector table at the address expected by the CPU, special handling for initialized data (segment in both RAM and ROM, with copy from ROM to RAM at startup), etc.

5. (6 pts) A development team discovers that their final program is too large for the available ROM. Briefly describe <u>three</u> things this team should do in their efforts to reduce the program size so it will fit.

First, note that we need to focus on ROM (not RAM) usage, so reducing the sizes of task stacks or kernel data structures won't help. What's in ROM? Code and initial values of initialized data segments. The question thus really boils down to what we can do to reduce our code size. Several ideas are given in the text on pages 255-257, of which the most likely to pay dividends (in my opinion) are:

- Removing unused RTOS functions from the program, if any exist
- Making sure that only functions you actually use from a library are included
- Using efficient algorithms and making sure compiler output for all C constructs is reasonable
- Wherever possible, eliminating variables larger than the word size (not to save data space in RAM, but to reduce the number of instructions in ROM, since code manipulating larger values can require many more instructions)

6. (40 pts) Mark each of the following true or false by circling **T** or **F** respectively.
a. The entries in each message queue managed by YAK are of a user-defined type.

False. The entries in the queue that YAK manages are *always* void pointers. Those void pointers may in turn point to something else, but as the YAK specification and the class slides makes clear, the actual queue is always an array of void pointers.

b. With timer tick interrupts every 20ms, a task that delays itself for 2 ticks might actually run again in just 23ms.

True. If the task calls delay just before the next clock tick, it will be made ready in just a little over one clock period.

c. A timer callback function will always run using the stack of the task that originally called the function.

False. Our text describes two ways in which timer callback functions might be implemented: the function could be called from a special ISR when the timer expires, or a special task could cause the function to run. There is therefore no reason to assume that the task will execute on the task of the original task that called the function.

d. One cause of clock jitter is that tasks do not always run immediately after being added to the ready list.

True. Jitter is undesirable variability in the period of a task: it is supposed to run very consistently, but doesn't. Variability in a task that is, say, supposed to run every 10 clock ticks does not arise because the heartbeat timer is inconsistent, or because the tick handler is inconsistent in putting the task in the ready list. It arises because a task has variable delay after it is marked ready, depending on the priorities of other ready tasks and on what interrupts occur.

e. The heartbeat timer is always the most accurate timer in an embedded system.

False. Many systems use separate hardware timers when very accurate timing is needed. Consider this quote from page 187 of the text: “It is not uncommon to design an embedded system that uses dedicated timers for a few accurate timings and uses the RTOS functions for the many other timings than need not be so accurate.” Note that the RTOS functions referred to here are based on the heartbeat timer.

f. In YAK, the kernel always resets events in an event group after they occur.

False. In YAK code that uses events, all setting and resetting of events are the responsibility of the application code. See the class slides, application code for Lab 7, and Figure 7.8 in the text.

g. Many RTOSs offer fast and predictable functions that allocate and free variable-size blocks of memory.

False. Allocating and freeing variable-sized blocks is the service that malloc and free provide, and their overhead and unpredictable execution times make them unattractive in embedded systems with real-time constraints. Many RTOSs offer a simplified but still useful service: routines that allocate and free *fixed-size* memory blocks. Routines that manage a pool of fixed-size blocks are much simpler, so they can be both fast and predictable. See page 196.

h. In our ISRs, the call to YKEnterISR constitutes *fair warning* to YAK that interrupt code is running.

True. This is the terminology that our text uses in Section 7.5.

i. It is a good design principle to have separate tasks to take actions in response to separate stimuli.

True. This is one of the recommendations from our author on page 229.

j. The encapsulation of a message queue is possible only if the queue itself is stored on a stack.

False. Encapsulation is the hiding of implementation details, not forcing the data structures to be completely local. In any event, a queue allocated on a task stack is problematic, since the whole idea is to allow information to be shared between tasks, or between an ISR and a task.

k. Time slicing can be helpful in decreasing critical task response time.

False. Time slicing is alternating the execution of tasks during short execution intervals (between heartbeat timer ticks, say). This is useful when humans want to see all current jobs make progress, but it just adds overhead and slows everything down in a real-time system. See page 232.

l. If a real-time system uses *rate-monotonic* scheduling, no deadlines will be missed.

False. Rate-monotonic scheduling assumes that task priorities are assigned based on how frequently those tasks will run. This can certainly be helpful but it is not a silver bullet. Deadlines are guaranteed to be met only if the CPU utilization stays below about 70%. Regardless of how you assign task priorities, you can still have more to do than the CPU can handle.

m. Many RTOSs allow you to save memory by removing services you do not use.

True. See Section 8.6 in the text.

n. In one common power-saving CPU mode, DMA transfers can continue while the processor sleeps.

True. In this particular mode, the CPU is inactive but on-chip peripherals continue to function, including DMA channels. See page 258.

o. The contents of ROM may include *shadow segments* with initial variable values.

True. The shadow segments (as illustrated in Figure 9.5) are copied to RAM at startup time, and then all runtime references to the variables will access the copy in RAM. See pages 268-273 in the text.

p. A monitor is a program that resides on the target and knows how to load new programs onto the system.

True. See pages 280 and 323 in the text.

q. To use assert macros on your target, you may have to write a custom routine to stop normal execution.

True. Think about what assert() does on desktop systems. If the condition isn't satisfied, execution terminates with an explanation. The assert mechanism is still very useful on an embedded system, but you can't just dump the explanation to the screen (since there is usually no screen) and call exit(). Depending on what you have to work with, you may have to write you own "bad_assertion" function to stop execution and somehow convey what went wrong. See the discussion on page 306 and 307.

r. The USB driver for the HP Inkjet printer (discussed in class) was implemented as a separate task.

False. The designers considered making it a task, but ruled it out because it would cause too many context switches. They ended up implementing it as a library routine, which had some important implications regarding the method they ended up using to enforce critical sections. See the class slides.

s. The lack of mutual exclusion on shared data accesses was an important factor in the Therac-25 accidents.

Sad but true. Unfortunately, the designer was writing multi-tasking real-time code that would have benefited from an RTOS, but he apparently lacked the background to create and use reliable semaphores. His design occasionally failed, and the result was deadly. See the Leveson paper.

t. Prof. Edward Lee argued that core abstractions in computing must be changed to reflect the passage of time.

True. This is the main point of his article that we discussed. He noted that the fundamentals of computing are about the transformation of data independent of considerations of time. For "cyber-physical" systems that interact with physical processes in the real world, precise timing of actions is critical. Integrating the notion of time into computing at a very fundamental level would affect systems at almost all levels (hardware and software) and help address some of the biggest challenges we face in designing complex embedded systems.

7. (6 pts) What is *priority inversion* and what consequences can it have?

Priority inversion is a temporary inconsistency between numerical task priorities and true dynamic importance of what should run next. In other words, the most important task to run right now is not necessarily the ready task with the highest priority. The classic example of potentially nasty consequences of this can be found in the slides and in the book in Figure 6.17. Note that this scenario takes a semaphore and three different tasks. The consequences arising in this situation can be missed deadlines and possible system failure (in hard real-time systems).

While even a scenario with just two tasks can exhibit a form of priority inversion, if the code is well written, this situation is much less likely to have similar dire consequences. The worst-case scenario, of course, is when the lower priority task takes a semaphore and then the higher priority task is made ready and tries to obtain the semaphore and blocks. In this situation, certainly the highest priority task is delayed, but that's a long way from breaking the system. Simply put, if this situation causes your system to malfunction, it is poorly designed. If you wrote the code, you would surely know that both tasks take the same semaphore, so the possibility that the higher priority task would have to wait would not be a surprise. You would make sure that there was enough time to run the critical section in the lower priority task without causing the higher priority task to miss a deadline. To do anything less would be negligence. In contrast, the priority inversion scenario with the three tasks is more difficult to see coming and more difficult to analyze. To avoid this situation, it is important to get some kind of RTOS support, such as priority inheritance on the semaphores.

8. (6 pts) Assuming a YAK-like RTOS, what are the important differences between *semaphores* and *events*?

Semaphores can be used for mutual exclusion and for signaling, while events only serve for signaling. Events have a little more overhead, and their functionality differs in a couple of key ways: a task can block on (and hence be awakened by) multiple events at the same time (not true of semaphores) and a single event can unblock multiple tasks, whereas tasks can block on single semaphores only, and at most one task is unblocked by a semaphore post. Finally, you have to do explicit resets with events, and no such operation is required with semaphores.

9. (6 pts) Section 10.1 of our text describes a methodology for testing that requires the creation of *test scaffold code*. Briefly describe the motivation for this approach, how it works, and its limitations.

The motivation for this approach is simply to test as much of the code as possible on the host (as opposed to the target). To use this approach, the designer starts early in the design phase by making a clean interface between hardware-independent and hardware-dependent parts of the code. The test scaffold code is created to replace the hardware-dependent code in the version you run on the host. The scaffold code mimics hardware events, often using a simple scripting mechanism, and those events trigger actions by the hardware-independent code. In this form of testing, you are making sure that your hardware-independent code is doing the right thing in every case. Two important limitations of this approach are that the hardware-dependent code cannot be tested, and the execution timing does not reflect what will actually take place on the target.