ECEn 425 Fall 2015
Midterm 2 Solution

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

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

void main(void) {
    YKInitialize();
    YKNewTask(Task1,(void *)&Stk1[SSIZE],10);
    YKNewTask(Task2,(void *)&Stk2[SSIZE],15);
    YKNewTask(Task3,(void *)&Stk3[SSIZE],20);
    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) {
        YKDelayTask(5);
       /* misc. code, no output */
    }
}

void Task2(void) {
    printString("D");
    YKNewTask(Task4,(void *)&Stk4[SSIZE],25);
    printString("E");
    YKSemPend(SemA);
    printString("F");
    while (1) {
        YKDelayTask(5);
       /* 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) {
        YKDelayTask(5);
       /* misc. code, no output */
    }
}

void Task4(void) {
    printString("J");
    YKEventSet(Ev1,FLAG1|FLAG2);
    printString("K");
    YKSemPost(SemA);
    printString("L");
    while (1) {
        YKDelayTask(5);
       /* misc. code, no output */
    }
}
```

PROGRAM OUTPUT:

The program outputs **ADEGJBHCIKFL.** 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.

2. (5 pts) What are *timer callback functions*, why are they useful, and how might they be implemented?

Timer callback functions are a nifty RTOS service described in the book on pages 188-191. Basically, you ask the system to call a particular function after a specified time period has elapsed. This is useful because it can really simplify task code when a sequence of actions must be taken with precise timing. As we discussed in class, the overall mechanism can be based on heartbeat timer ticks (and hence with timing that is not terribly precise) or it could be based on hardware timers. In either case, the actual function could be called by an interrupt handler (the timer expiration handler, or the tick handler) or the ISR could cause a special high priority task to call the appropriate function. Note that, in either case, the function will not run on the stack (in the context) of the task that originated the call. Note also that the caller did not block – this is an important point that results in the simplification of the code in the originating task.

3. (10 pts) Circle each function below whose normal implementation includes a call to the scheduler.

[✓ YKDelayTask] YKQCreate [✓ YKNewTask] [✓ YKEventSet] [✓ YKEventPend]
[✓ YKQPend] [✓ YKSemPost] YKInitialize YKSemCreate YKEventReset

The YAK specs require a call to the scheduler whenever the status of a task might change, so that preemptive scheduling will work as the user expects. Therefore functions that include a call to the scheduler in their source code are: YKDelayTask, both pend functions, the post function (and the logically similar YKEventSet), and YKNewTask. Functions that do not include a call to the scheduler include both create functions (the caller's state will not change, nor can any task be waiting for the creation), YKInitialize (which runs before any calls to the scheduler), and YKEventReset (which clears bits in an event group but does not unblock any tasks).

4. (10 pts) Circle each function below whose execution can cause a task to be added to the ready list.

YKDelayTask	YKQCreate	[✓ YKNewTask]	[✓ YKEventSet]	YKEventPend
YKQPend	[✓ YKSemPost]	[✓ YKInitialize]	YKSemCreate	YKEventReset

Some of these remove tasks from the ready list rather than put one in. This is true of YKDelayTask, YKEventPend, and YKQPend, all of which can remove the calling task from the ready list. YKQCreate and YKSemCreate don't affect the status of any tasks, so they definitely don't put tasks in the ready list. YKSemPost can certainly unblock a task and cause its TCB to be added to the ready list. YKNewTask will put the newly created task in the ready list – tasks are always assumed to be in the READY state when created. According to the YAK specs, YKInitialize must create the idle task, so it causes a task to be added to the ready list. YKEventSet is basically the post function for events, as it can cause multiple tasks to unblock and therefore be added to the ready list. That leaves YKEventReset. According to the YAK specs, this function does not cause tasks to unblock, so it should not be circled.

5. (4 pts) What are the two rules that ISRs in an RTOS environment must follow that do not apply to task code?

See pages 199-205 in the text. Rule 1: An interrupt routine must not call any RTOS function that might block the caller. Rule 2: An interrupt routine may not call any RTOS function that might cause the RTOS to switch tasks unless the RTOS knows that interrupt code (and not task code) is running.

6. (40 pts) Mark each of the following true or false by circling **T** or **F** respectively.

a. [✓ **T**] **F** Some of the best candidates for encapsulation are accesses to shared data.

True. See the class slides.

b. [✓ **T**] **F** Studies show as much as 3x difference in productivity based on how often programmers are interrupted.

True. See the class slides. This is important to keep in mind when you are out there in the workplace. Control your environment to the extent you can.

c. **T** [✓ **F**] The RTOS is aware of which shared resource a particular semaphore protects.

This is false. See page 162.

d. **T** [✓ **F**] Clock jitter arises because heartbeat timer interrupts are not asserted at precise intervals.

False. Timer ticks are generated very accurately. The problem is that you don't know what else is running in any short interval, so you don't know how long it will take for the task to actually get some CPU time. See the class slides.

e. [✓ **T**] **F** It is common for embedded microprocessors to contain timers unrelated to the heartbeat timer.

True. See page 186 in the text.

f. [✓ **T**] **F** In YAK, kernel code never dereferences the void pointers in a message queue.

True. When we dereference a pointer, we access what the pointer is pointing at. Although each entry in a message queue in YAK is defined to be a void pointer, the actual contents could be other data cast as a void pointer. In any event, the kernel just reads and writes entries in the queue – it never needs to access what the pointers are pointing at (if they are pointers).

g. [✓ **T**] **F** In a rate monotonic system, the task assigned the highest priority is the task expected to run most frequently.

True. See class slides.

h. **T** [✓ **F**] In any real-time system with rate monotonic priority assignment, all deadlines are guaranteed to be met.

False. It is not a silver bullet. Assuming rate monotonic priority assignment (assigning task priorities in the order of frequency of events to which they must respond), deadlines are guaranteed to be met only if the CPU utilization stays below 70%. Regardless of how you assign task priorities, you can still have more to do than the CPU can handle.

i. T [✓ F] A nonconforming interrupt routine does not save or restore register context when it runs.

False. See problem 9 at the end of Chapter 7 – that we discussed in class. Note that an ISR that didn't save context would pretty much break the system – anything it interrupted would malfunction. The label "non-conforming" in this sense means that it doesn't notify the RTOS that interrupt code is running, so it is limited in what kernel functions it can call and it must not allow interrupt nesting.

j. T [✓ F] In most embedded systems, a loader places the program in memory and performs all required address fix ups.

False. There is no loader in embedded systems and no fix up is required – the linker/locator determines the final addresses. See page 267 in the text.

k. T [✓ F] In all common power-saving modes, suspended processors require a reset to start up again.

False. In many cases an interrupt is sufficient. See page 258 in the text.

l. [✓ T] F A monitor is a program that resides on the target that can receive and run new programs on the target.

True. See pages 280 and 323 of the text for a discussion of this overloaded term.

m. [✓ T] F 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 the target, but you can't just dump the explanation to the 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 pages 306-307.

n. [✓ T] F Increasing the number of tasks is likely to increase the need for semaphores and message queues.

True. See page 222. The more you break things down, the more communication and synchronization you are likely to need.

o. [✓ T] F Instruction set simulators can help determine response time and throughput in application code.

True. See page 302. While the simulator will not run at the same speed as the target, it can provide execution statistics that (coupled with other information about the target) can allow timing on the target to be determined.

p. [✓ T] F Lockheed-Martin's space shuttle code averaged less than 1 error per 100,000 lines of source code.

True, and absolutely astonishing, given that typical rates are 60 errors per 1000 lines. See the class slides.

q. [✓ T] F The lack of mutual exclusion on shared data accesses was an important factor in the Therac-25 accidents.

Sad but true. See the Leveson paper. 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.

r. [✓ T] F Some trading firms have server farms near stock exchanges to reduce the delay for their electronic orders.

True. See the article and video on high-frequency trading.

s. [✓ T] F 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.

t. **T** [✓ **F**] The paper on memory tests suggested separate tests for data, address, and control lines.

False. The author points out that it is not possible to develop a general test for control line problems, given the implementation differences between platforms. They simply aren't standardized like address and control lines. His first two tests focus on data and address lines, but his third test is a device test. (Fortunately, problems with control lines will almost certainly be detected in the course of these three tests, so wiring problems will not go unnoticed.)

7. (5 pts) What particular challenge does *initialized data* present in embedded systems, how is it dealt with, and why does the challenge not arise in conventional (desktop) systems?

Initialized data refers to global variables in the C source code that are assigned a value when they are declared. These are easily dealt with in conventional tool chains, because that initial value will be part of the memory image for the program, and so it will always have the correct value at the beginning of execution because the program will be loaded into memory from disk with that value already there (and zero in all uninitialized locations).

Why is the situation different in embedded systems? The initial value must be in ROM (where else could it be?) and it must be in RAM at runtime because it is, after all, a variable. The problem is that there is no loader to automatically move the value (in the way that a loader copies the binary into memory in desktop systems) when the system powers up or resets. The problem is addressed by creating two separate memory slots for the initialized data segment – one in ROM and one in RAM. Then, code must be added to copy the "shadow" segment from ROM into RAM, and this must take place before the execution of any normal code that accesses initialized variables. So what code can do the copy? Recall that the first code to run in systems with an RTOS is main() in the user code. In YAK code, it would probably make sense to add this copying to the responsibilities of YKInitialize, because the system can help you with the nitty-gritty details, but you need to pass it control since kernel routines run only when called.

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

Semaphores can be used for mutual exclusion and for signaling, while events only serve for signaling. Events have a bit 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. (10 pts) Identify and briefly describe <u>five</u> specific memory segments (by category) that necessarily are handled differently by the linker/locator.

The key here is understanding that segments are groupings of similar memory contents that must receive special consideration by the locator, particularly regarding their placement in the target system's memory space. The question pertains to segment handling in embedded systems because we're considering the actions of a "linker/locator" (see page 263). We can start out with (1) a segment for regular code that must be placed in ROM. Another segment (2) would be a starting code segment that runs when the system turns on, placed in memory at the address where the CPU starts execution when powered-up. On the data side, we need (3) a segment for uninitialized data that resides in RAM and need not have any space allocated in ROM. We also need (4) a segment for initialized data, since it resides in RAM but must have a shadow segment in ROM that contains its initial values. Finally, (5) segments for string constants can be problematic and require handling separate from other data since C allows those string "constants" to be modified. Some of you came up with other possibilities that were also acceptable. Note, however, that a stack segment is really the same as uninitialized data – in fact in our application code each stack is just a global array in the application code. (In fact, the compiler has no idea this array will be used for a stack.) The key is to identify different logical divisions that require different handling (e.g., placement, copying) by the locator.

(1) Regular Code that must be placed in ROM
(2) Starting Code that runs when the system turns on
(3) Uninitialized data that resides in RAM
(4) Initialized data because it needs RAM and ROM
(5) String Constants