ECEn 425 Fall 2017
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],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. (10 pts) Circle each function below whose normal implementation includes a call to the scheduler in its source code.

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 call 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).

3. (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 certainly 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.

4. (6 pts) Explain why a failed assertion can save many hours of debugging.

As I see it, there are really two keys here. The first is that a failed assertion tells you something is wrong, and the second is that you have a precise point in the code's execution where a bad value was detected. Sometimes code malfunctions and we never even see it because the system never "fails" in an obvious sense, so the first point is quite important. Once you know something is wrong, normal debugging often involves a great deal of guesswork, experiments, and backtracking to find the actual cause of the system's misbehavior. The failed assertion does not necessarily give you the precise point of the error, but you are probably much closer to it than you'd be without the assertion. Chances are the assertion fails on parameter values for a function, say, and you still have to work your way back up the call chain to figure out why the parameter value is out-of-range, but this is often just a small fraction of the work required to figure out what went wrong without the assertion.

5. (6 pts) One expert argued that errors are more common and pervasive in software than in other technologies. What is fundamentally different about software?

In the context of our class, this is an important issue to reflect on. Creating software is a very different thing than building a bridge, say. (How does one create software that exceeds specs by 50%?) I certainly didn't expect you to provide all the following, but here is some food for thought. (I think Parnas is to be credited with several of the following points.)

- Software is more complex than other technologies, since it has many interacting pieces. (Consider the size of a compact but complete description of a nontrivial software system.)
- Software components are connected in ways not possible in the physical world. (Changing a piece here might break a piece way over there – in a way not possible with the parts of a bridge, say.)
- Software is extraordinarily sensitive to small errors. (Even trivial typos can dramatically change the outcome, even more than fundamental oversights by the designer.)
- Software is hard to test. (Its functionality is highly discontinuous, so interpolation between tested values can't help. Spot testing is inadequate, and exhaustive testing is impossible.)
- Software errors are highly correlated. (All software errors are design errors, and these are far from independent in a given program. A consequence here is that one cannot replicate software components to increase reliability.)
- Software exists in an abstract space where the laws of physics do not apply. (Designers can pretty much make it do whatever they want, which can make it very hard to understand and debug.)

The bottom line is this: the creation of software is very challenging! But Lockheed-Martin's example with the space shuttle code is evidence that it can be done.

6. (34 pts) Mark each of the following true or false by circling **T** or **F** respectively.

a. A segment is a division of a program that corresponds to all code or data in a single source file.

False. Segments are logical divisions of a program that are similar and that can be treated as a unit, such as "startup code" or "vector table contents" or "initialized data". There is no connection between segments in the final program and the makeup of the original source files for the program. (You wouldn't want the desire for modularity to affect the efficiency of the compiled result.)

b. According to our text, the overhead of writing to a queue typically exceeds that of creating a task.

False. You've coded these up – compare the complexity and runtime of YKQPost with YKNewTask. In the example timings in Table 8.1 (p. 223), creating a task is actually much higher than the other common RTOS operations listed – by quite a lot. In well designed code, you'd want to pay the price for this overhead just once per task when the system starts up. (It wouldn't make a lot of sense to repeatedly create and then destroy or delete the same task.)

c. Operator action within very narrow time windows was a critical factor in the Therac-25 accidents.

True. "The second lesson [learned] is that for complex interrupt-driven software, timing is of critical importance. In both of these situations, operators action within very narrow time-frame windows was necessary for the accidents to occur." (Quote from Miller, on p. 38 of Leveson, Turner article, also in class slides) As the assigned article points out, fundamental problems in the control software were made manifest by revisions to the user interface that made certain operations more convenient for the operator.

d. A 2014 analysis of the WinVote system revealed the use of hardwired passwords and wireless encryption keys.

True. See the assigned paper written by Epstein. Virginia's IT department conducted the security analysis in 2014 – at the time, the machines were still in use in some jurisdictions. Among the many problems they discovered were these: the system used a hardwired Windows administrative password (with no interface to change it) and a hardwired key used in the WEP wireless encryption (despite WEP being declared obsolete 10 years earlier for security flaws).

e. For best performance, code should be executed out of ROM since it is faster than RAM.

False. RAM is generally faster than ROM, so if anything, you copy the code to RAM on startup (in a manner similar to shadow segments for initialized data) and execute out of RAM. See the discussion in the text beginning on page 274.

f. An RTOS scheduler was discussed in class that determines the highest-priority ready task in constant time.

True. We discussed this rather nifty characteristic of µC/OS in class. See the class slides. (This fits into the category of having a clearly identifiable worst-case execution time, in this case the same as the best case execution time.)

g. In one common power-saving mode, a suspended processor will start up again on any interrupt.

True. See the class slides.

h. A locator map is likely to include the addresses of all local variables defined within task functions.

False. The locator does not concern itself with the contents of stack frames, and it certainly cannot determine the actual runtime memory addresses of local variables. These are determined by the order of execution and not known at the time the linker/locator is doing its work.

i. ROM emulators are a popular mechanism for getting software into the target system for debugging.

True. See page 277 in the text. The availability of inexpensive flash memory has probably reduced its use somewhat, but not all embedded systems use flash, and this is an effective approach during the development and debugging phase for such systems.

j. In a *rate monotonic* system, the task assigned the highest priority is the task expected to run most frequently.

True. See the class slides.

k. 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).

l. Each time the YAK kernel runs, the kernel's stack becomes the currently active stack.

False. The kernel does not have its own stack. It is not a separate task that runs; it is simply a collection of routines that are called when the application code needs kernel functionality, so the routines run on the stack of the calling task (or on the stack of the current task if called from interrupt code).

m. The USB driver for the HP Inkjet printer locked the scheduler during its critical sections.

True. See the class slides. The example is quite informative because it saves the interrupt status at the beginning, disables certain interrupt levels, locks out the scheduler (so you don't switch to something else and go for a long period of time with some interrupts disabled), does its thing, and then restores everything at the end. This is a good deal more complicated than what we have to do anywhere in our kernels, and it is important to understand why.

n. With timer tick interrupts every 10 ms, a task that delays itself for 2 ticks might actually run again in just 12 ms.

True. If the call to delay runs just before one clock tick, and not much is running when the tick occurs that makes the task ready, the task could be delayed for 12 ms.

o. If application code runs out of entries in queue, the RTOS will automatically increase the queue size.

False. Application programmers get no such help from the RTOS. Remember, it isn't managing memory to begin with, so it can't help much when you need some more.

p. A new $300 million cable reduces the signal delay between New York and London by a few milliseconds.

True. See the assigned paper by Schneider that discussed high-frequency trading. Beating everyone else (in responding to changes in the market) is so important that having a few extra milliseconds is apparently worth laying out a ridiculous amount of money for a slightly faster connection.

q. C functions `malloc` and `free` are rarely used in embedded systems, primarily because they waste memory.

False. We know that these functions are not widely used in embedded systems, but it is because they are *slow* and/or *unpredictable* in their execution times, not because they are inefficient in their use of memory. See section 7.4.

7. (8 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.

8. (8 pts) What is *encapsulation*? What are the best candidates for encapsulation in YAK application code? Why?

Encapsulation is the hiding of implementation details within a function or set of functions so that things are simplified for the callers of the function(s) in question. The hidden details could be the data structure used, the details of a hardware interface, the mutual exclusion approach taken within a critical section to avoid a shared data problem, etc. In YAK, the best candidates for encapsulation are probably routines that use message queues, semaphores, event groups, and shared data in general. In each of these cases, there are operational details that, if hidden from callers, could simplify the code and make its overall operation more reliable.

9. (8 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.