ECEn 425 Fall 2019
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:

This 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 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, but never move a task *into* 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. In contrast, 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 to 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.

3. (6 pts) What is *jitter* and what causes it?

From the book (p. 253) and the class slides (6:31), jitter is the difference between the programmer wanted the task to run and when it actually runs. (Alternately, it is the variability in the period of task, assuming that the task runs with a fixed frequency.) Jitter is therefore closely related to the variable delay between a task getting into the ready list and actually running. The major causes of jitter are the execution of ISRs and higher priority tasks, as well as inconsistencies that arise in using a delay mechanism based on a heartbeat timer. Note that jitter is NOT caused by inaccuracies in the heartbeat timer or inconsistency in getting the task into the ready list at the right time – measurements of these factors would be remarkably consistent in code using an RTOS.

4. (6 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 the initial value is present in the memory image for the program when it is loaded into memory from disk before the program runs. Thus, each memory location has the correct value at the beginning of execution – before a single instruction of the program is executed.

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 copy the initial value (in the way that a loader copies the initial memory image from disk in desktop systems) when the system powers up or resets. The problem is addressed by (1) creating two separate memory slots for the initialized data segment – one in ROM and one in RAM, and then (2) adding code to copy the "shadow" segment from ROM into RAM. Since the copy must take place *before* the execution of any normal program accesses to initialized variables, what code can perform the copy? Recall that the first code to run in systems with an RTOS is main() in the user code. Hence, in YAK code, it would probably make sense to add this copying to the other responsibilities of YKInitialize. The kernel can help you with the nitty-gritty details, but it can't run unless you call it, and it should be called very early on when the system boots or resets.

5. (40 pts) Mark each of the following true or false by circling **T** or **F** respectively.

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

True. A pointer is dereferenced when code accesses the thing the pointer points to. Entries in YAK message queues are defined to be void pointers, but the actual contents can be another data type cast as a void pointer, so any code that tries to use them as pointers would malfunction. In any event, the YAK kernel just reads and write entries in the queue – it never attempts to access what they point to (whether or not they are pointers).

b. **T F** 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.

c. **T F** In an RTOS, writing to a queue that is full typically blocks the caller if the caller is a task.

False. See page 176. The most common behavior in this case is to drop the message, as is done in YAK.

d. **T F** The task structure preferred by our text is an infinite loop that blocks in just one place.

True. See the discussion in the text on pages 229-231.

e. **T F** The use of packed data structures is likely to reduce data space while increasing code space.

True. Packing, say, multiple char variables into a single memory word can save space required to store data, but it is likely to require more instructions to access those values, in form of masking, shifting, etc. Hence, it is not clear that packed data structures result in overall memory savings.

f. **T F** Events are typically simpler and faster than semaphores.

False. Testing the values associated with events and unblocking multiple tasks has a little more overhead than semaphores. See page 192.

g. **T F** Code with a group of 3 events can always be rewritten with identical functionality using just 3 semaphores.

False. The two constructs are not always so easily interchangeable. As discussed in class, consider problem 7.2 2, where events in Figure 7.8 are to be replaced with semaphores). With several tasks using a mixture of WAIT_FOR_ANY and WAIT_FOR_ALL on subsets of the bits, you can construct examples with 3 event bits that cannot be matched in functionality with just 3 semaphores.

h. **T F** The effect of the call to YKEnterISR in each ISR is to disable the scheduler while the ISR runs.

True. The call to YKEnterISR is YAK's way of dealing with Rule 2 for interrupt code in an RTOS environment. (Rule 2: Interrupt code may not call RTOS functions that might cause a context switch unless the RTOS knows that an interrupt routine is running, and not a task.) The call to YKEnterISR provides fair warning to the kernel, so that all RTOS routines called while the interrupt routine is in progress will come back to the interrupt routine rather than allowing a task to run. "Essentially, this procedure disables the scheduler for the duration of the interrupt routine." (p. 205)

i. **T F** The scheduler should not run from the time an ISR starts until YKExitISR executes.

True. This is the desired consequence of ISR rule 2: no RTOS calls without fair warning. Neither the ISR nor its handlers nor kernel functions called by interrupt code should call the scheduler or task code will start running before the ISR is finished. See section 7.5.

j. **T F** Code written using static local variables can be faster than equivalent code using variables on the stack.

True. See page 256: "Many microprocessors can read and write static variables using fewer instructions than they do for stack variables."

k. **T F** For best performance, code should be executed out of ROM since it is faster than RAM.

False. RAM is generally faster than ROM, so best performance is obtained by copying the code from ROM to RAM on startup (in a manner similar to shadow segments for initialized data) and then executing the program in RAM. See the discussion in the text beginning on page 274.

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

True. In a rate monotonic system, tasks are assigned priorities based on their execution frequency. This has the advantage of giving a clear-cut and defensible way of assigning task priorities, and it has been proven that rate monotonic systems with utilization below a particular threshold can meet all deadlines. See the class slides.

m. **T F** 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 cannot determine the actual runtime memory addresses of local variables. These are determined by the order of execution and they may vary from function call to function call.

n. **T F** In one common power-saving mode, a suspended processor will start up again on any interrupt.

True. This is not the most aggressive of power-saving modes, but it might be one of the easiest to use. In this mode, the processor stops executing instructions, but on-board peripherals continue to operate, and they are able to awaken the processor via the standard interrupt mechanism when anything occurs that needs its attention. See the class slides.

o. **T F** A segment is a division of a program that corresponds to all code and 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. (We wouldn't want the desire for modularity to affect the efficiency of the compiled result.)

p. **T F** In µC/OS, the code to place a task in the "ready list" runs in constant time regardless of the task's priority.

True. This was noted in the class slides used in our discussion of the scheduling mechanism in µC/OS that offers some nice performance advantages because of its predictable execution time. Moving a task into the ready list and finding the highest priority ready task execute in constant time. Removing a task from the ready list is near constant time (there is an if statement in the C code).

q. **T F** In normal usage, an `assert` macro will compile to no code on systems shipped to customers.

True. In normal usage, assert macros test the specified condition and terminate execution (with some explanation) if it does not hold. This is very useful during development, testing, and debugging, but it is not useful (or even desirable) in shipped systems because information at that level is not useful to the end user. For this reason, the macro definition makes it very easy to change one #define that turns all asserts into nothing. The bottom line is this: they are still there in the source code (to help you debug the next round of changes), but no trace of them is found in the binaries in shipped products.

r. **T F** The USB driver for the HP Inkjet printer (discussed in class) disabled all interrupts during critical sections.

False. The designers disabled only those interrupts below the priority level of interrupts involved with the USB device. They didn't want to delay the servicing of higher-priority interrupts. See the class slides.

s. **T F** The Therac-25 software included multiple tasks, interrupt code, and a scheduler.

True. See the class slides and the Levenson paper. The point is that this was (or it should have been) very much the sort of application code we're addressing in this class. Think about it: lives could have been saved if *you* had written the Therac-25 code.

t. **T F** The Software Engineering Institute claims that typical software has about 6 bugs per 1000 lines of code.

False. As stated in the class slides, the actual estimate is 60 bugs per 1000 lines, an order of magnitude higher. This is an astonishing high number! Either almost all programmers are inept, or writing software is hard. I hope after taking 425 you will agree that it is the latter.

6. (10 pts) With RTOS code, the application programmer must determine how big each stack should be. Circle the letters of those factors below that do NOT in any way affect the size of the stack required for task T.

a. The number of other tasks in the system.
b. Local variables in functions called by T.
c. The number of distinct functions T call directly.
d. The number and size of registers in the CPU.

e. Global data references made in T's code.
f. The number and type of local variables defined in T.
g. The number of hardware interrupt levels.
h. The total number of semaphores in the system.
i. The number of distinct places in its code that T can block.
j. The length of the longest critical section in T's code.

Let's review how task stacks are used when an RTOS executes. The stack is used to allocate local (non-static) variables defined within the C function associated with that task, or within any function called by that task. It is also used by each ISR to save all registers when an interrupt occurs, so the storage required for a saved context is determined by the size and number of registers in the processor. Also, the number of interrupt levels supported by the hardware determines the worst case interrupt nesting and hence the total number of ISR-saved contexts that could potentially be saved on the stack at the same time. Thus, the letters that correspond to factors that DO affect the stack size are b, d, f, and g. Let's consider in turn each of the remaining alternatives (that you should have marked since they do NOT affect stack size). (a) Although it is an important factor in the total memory required for task stacks (since each task needs its own stack), the number of other tasks in the system does not affect how much space must be allocated for T's stack. (c) The memory required for a task stack is affected by the maximum number of functions that might simultaneously be active (through a sequence of function calls), but it does NOT matter how many distinct functions T calls directly. In other words, it doesn't matter if T calls 10 functions or 20 functions, so long as there is stack space for the worst-case memory requirements of those functions. (e) Global variables are not stored on the stack, and the stack is not used in any way to reference them, so global data references do not affect stack sized. (h) The RTOS maintains a block of memory associated with each semaphore (to track its availability and which tasks are blocked on it), but that memory is not on the stack. (i) Whether task T just blocks in one place in its code (a structure recommended by our text) or in 10 places in its code, there are no implications for the size of the stack required for that task. (j) It is important to recognize critical sections in our code and to use an appropriate method to enforce mutual exclusion (such as disabling interrupts at the start and enabling them at the end), but code in critical sections does not differ from normal code in how the stack is used. Summary: you should have marked a, c, e, h, i, j.

7. (6 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. Using this mechanism, task code can ask the kernel to call a particular function after a specified time period has elapsed. The task itself does not block, and it will NOT be involved in the later execution of the specified function – it is handing off to the kernel the details of actually making that function call and making it at the right time. As a consequence, task code can be dramatically simplified when a sequence of actions must be taken with precise timing. As discussed in class, the overall mechanism could be based on heartbeat timer ticks (and hence with relatively imprecise timing) or it could be based on a hardware timer. In either case, the actual function call could be made 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.

8. (6 pts) Why do embedded developers typically avoid using `malloc` and `free`, and what do they use instead?

See the text in Section 7.4. Calls to malloc and free are typically slow and – arguably a bigger problem – they have unpredictable (inconsistent) execution times. Instead, designers use RTOS functions that allocate and free buffers of a fixed size. (They are RTOS functions so that tasks requesting a block can be blocked until one becomes available.) The management overhead for fixed-size blocks is much lower than for blocks of arbitrary size (which complicates malloc and free), so these RTOS functions run quickly and have predictable execution times.

9. (6 pts) Briefly describe two separate methods that allow embedded-system software to be tested on the *host*, as opposed to the *target*. Detail the limitations of each approach.

Chapter 10 of our text describes two such methods: (1) using a simulator of the target machine (that models the execution of all instructions, and all the hardware including peripherals at some reasonable level), and (2) separating your application code cleanly into hardware dependent and hardware independent portions (with a clean functional interface between the two parts) and then replacing the hardware dependent portion with scaffolding code that mimics hardware operation at a high level and that is driven by scripts. The limitations of the simulator approach are that a commercial tool is unlikely to simulate all of your custom hardware, and that the interface and overhead of a simulator makes it difficult to do the extensive testing required to discover shared-data bugs (that arise when interrupts occur at just the right or wrong point in time). The limitation of the scaffolding approach are that you aren't testing all the code that will run on the target, and for the code that *is* running you've dramatically changed the timing. This makes it much harder to discover shared-data problems, and to say anything definitive about response and throughput of the application code.