ECEn 425 Fall 2016
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],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) {
        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 **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.

2. (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 to extend 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 things will malfunction. 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 would 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 particular 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.

3. (10 pts) Circle each function below that can be safely 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 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. Thus, you should have circled YKDelayTask, YKNewTask, YKEventSet, YKEventPend, YKQPend, and YKSemPost.

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 adjacent 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) 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 execution times. Instead, designers use RTOS functions that allocate and free buffers of a fixed size. The management overhead for fixed-size blocks is much lower than for blocks of arbitrary sized blocks, so these RTOS functions run quickly and have predictable execution times.

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

a. **T F** The construct "((void *) 1)" is interpreted as a void pointer to a location containing the value 1.

False. The expression takes the integer value 1 and casts it as a void pointer, so this ends up being a void pointer with the numerical value 1 (as if it were pointing to the byte at address 0x0001). Obviously, it would not be a good idea for the program to dereference a pointer constructed in this way! However, if you want to pass a message with the integer value 1 through a queue that expects void pointers, this is what you'd do. Compare the code in Figures 7.3 and 8.2

b. **T F** On some platforms, using static variables instead of stack-based variables can reduce code size.

True. This obviously depends on the instruction set architecture and addressing modes the CPU supports, but this is one of the possible memory-saving optimizations that our book mentions specifically. See page 256.

c. **T F** High-frequency trading is considered a cause of the Flash Crash, a 9% drop in the stock market in May 2010.

True. See the assigned paper written by Savani. As an investor, it would be pretty scary to see the value of your investments fall by 9% in just 20 minutes! In complex systems, there are almost always multiple factors in play, but it is widely believe that one of the root causes was HFT.

d. **T F** A 2014 analysis of the WinVote system revealed the use of obsolete wireless encryption and an unpatched OS.

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 an old version of Windows XP Embedded with no patches installed since 2004, and the system used WEP wireless encryption which was declared obsolete in 2004 due to security flaws.

e. **T F** Some embedded systems execute with programs stored in RAM rather than ROM.

True. RAM is generally faster than ROM, so some faster processors benefit by copying the code from ROM to RAM on startup (in a manner similar to shadow segments for initialized data) and then executing out of RAM. See the discussion in the text beginning on page 274.

f. **T F** If a message queue is to be encapsulated correctly, the queue itself must be stored on a task 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 multiple tasks (or an ISR and a task) to share information.

g. **T F** ISO 9000 certification requires an outside audit of a company's entire development process.

True. See the class slides (including those featuring Dilbert). Certifications like this are obviously a big deal and not to be taken lightly. Adopting a consistent development process is critical for any successful firm.

h. **T F** 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.

i. **T F** A *ROM emulator* is a software module used within an instruction set simulator to model memory.

False. A ROM emulator is a hardware device that plugs into a ROM socket on the target system. It also has a connection to the host, so the contents of memory can be modified very easily (with far less overhead than swapping out the memory chip).

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

False. This approach can certainly be helpful but 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.

k. **T F** Prof. Edward Lee argued that common computing abstractions have failed because they hide timing details.

True. The abstractions he mentions specifically include the instruction set architecture, the programming language, the operating system, and the network. All of these provide an interface that hides properties and details that affect timing, yet embedded systems (and cyber-physical systems in general) need to take actions in a precisely timed way. The conflict suggests that we need a major rethink of the fundamentals in computing systems.

l. **T F** Simplistic memory tests may fail to detect that memory chips have been removed from the board.

True. This point is made by Barr in the article on memory testing that was assigned. It can happen because of the capacitive nature of unconnected wires. As a consequence, a good test routine will not consist solely of reading from the location just written to.

m. **T F** The USB driver for the HP Inkjet printer was implemented as a separate task.

False. See the class slides. 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.

n. **T F** A *timer callback function* will always run using the stack of the task that 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 function will execute on the stack of the original task that initiated the timer callback function in the first place.

o. **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. For this reason, the macro definition makes it very easy to change one #define that turns all asserts into nothing – 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.

p. **T F** Part of a *monitor* is a program running on the host that provides a debugging interface to the target.

True. See pages 280 and 323 of the text for a discussion of this overloaded term. The monitor combines functionality on the host (including a debugging interface) with code on the target (that supports downloading new software, putting it in RAM and then running it).

q. **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 the end of problem 2 at the end of Chapter 7 (replacing events in Figure 7.8 with semaphores). With several tasks using a rich mixture of WAIT_FOR_ANY and WAIT_FOR_ALL on subsets of the bits, you can construct examples with 3 event bits that can't be matched in functionality with just 3 semaphores.

7. (6 pts) Briefly describe similarities between the software structure in the Therac-25 and the software that has been our focus this semester in this class.

According to a sidebar in the Leveson and Turner article (and our discussion in class), the Therac-25 software included a preemptive scheduler, at least three "critical" tasks, at least seven "non-critical" tasks, and several interrupt routines. The combination of task and interrupt code sounds a lot like applications developed for RTOS-based systems. In fact, the Therac-25 code exhibited the same shared-data and synchronization problems that commonly bite embedded developers, but the consequences were horrendous.

8. (8 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 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 each 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.

9. (10 pts) After coding the last module, you observe that your program is too large for the ROM your boss wants to use. Briefly describe the three things you could do that are most likely to reduce your program size so that it will fit.

First, note that we need to focus on ROM (not RAM) usage, so reducing task stacks 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
- 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 (since code manipulating larger values can require many more instructions, not because it saves data space)