UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Midterm Examination

Instructor:	B. F. Cockburn
Exam date:	February 28, 2018
Exam duration:	50 minutes
Aids permitted:	A paper or electronic copy of the course lecture slides can be freely consulted. Electronic lecture slides can be consulted on a tablet or notebook computer. The lecture slides can be annotated with the student's comments. No other references, including model solutions, can be consulted. The Internet cannot be accessed with any device. Electronic calculators of any kind are permitted.

Instructions:

1. Please enter your printed name, signature and I.D. number on this page.
2. Verify that this booklet contains 6 pages (including the cover page).
3. Neatly enter your answers in the spaces provided.
4. Use the reverse sides of the pages for rough work.
5. Take into account the marks per question when budgeting your time.

Student name: _____**Model**_____**,**___**Solution**_____
Family name **Given name**

Signature: _____

Student I.D.: _____

Question	Time	Worth	Mark	Subject
1.	10	20		Basic Concepts
2.	15	30		Software in Embedded Systems
3.	12	25		MicroC/OS Implementation
4.	13	25		Communication Interfaces
Total	**50 mins**	**100**		—

Question #1 (Basic Concepts)

Carefully explain the similarities and differences among the following pairs of concepts:

(a) In the C programming language, the **`(void *)`** type vs. the **`(int *)`** type

[3 marks] A (void *) is a typeless pointer. Runtime type checking is turned off for pointers of this type. Pointers of any other type can be assigned to a (void *). A (void *) cannot be dereferenced.

[3 marks] An (int *) is a pointer to a single-precision integer variable. Type-checking is performed when a value of this type is assigned to a variable or passed as a parameter to a function. A pointer of type (int *) can be cast or simply assigned to a (void *).

In C compilers it is common for all pointers to be implemented as unsigned single-precision integer values.

(b) Buffer overflow vs. stack overflow

[4 marks] Buffer overflow occurs when a write pointer, which is being used to add data items into a buffer, goes beyond one end of the buffer. This means that data is being written outside the buffer, possibly overwriting data and/or code. Malicious code can also be written to memory in this way.

[3 marks] Stack overflow is a type of buffer overflow when the one top-of-stack pointer goes beyond the fixed size limit in memory that was established when the stack was allocated memory space. Memory locations beyond the stack boundary are overwritten, damaging data and/or code.

(c) Fast response time vs. deterministic response time

[3 marks] Fast response time is when there is a relatively short interval from the time that (usually external) event occurs, and when an embedded system takes appropriate action.

[4 marks] Deterministic response time is when the response times have relatively small variability. The actual response time may be short or long; when it is deterministic the time is relatively constant and hence predictable.

Question #2 (Software in Embedded Systems)

Consider the following three alternative multitasking software architectures: (1) round robin time slicing, (2) periodically scheduled state-driven code, and (3) preemptive multitasking.

(a) Assuming that the only possible interrupts in the embedded system are timer-triggered interrupts, rank the three software architectures (from best to worst) with respect to how well the embedded system designer can ensure that all tasks are allocated a predictable fraction of the CPU's execution time. Carefully justify your answer and state any reasonable assumptions that you are making to determine your ranking.

[5 marks] (1) Round-robin timing slicing uses interrupts from a hardware timer to ensure that the active tasks are given accurately defined slices of time for execution on the CPU. Thus the allocated fraction of the CPU's execution time is accurately controlled and predictable.

[5 marks] (2) Periodically scheduled state-driven code allows the CPU's time to be allocated accurately to the different tasks on different scheduling lists provided that the repetition periods are harmonically related, the starting times of the lists are interleaved properly to avoid running into each other, and there is plenty of idle time so that the execution times of the tasks on different lists never conflict. However, the execution times of the different task states is not easily to determine; as well, additional analysis is required to determine how frequently each state is executed for each task.

[5 marks] (3) Pre-emptive multitasking gives the highest priority tasks control over their own execution time, subject to how much time they spend being blocked on conditions. But it is hard to predict exactly how much time a task, even a high-priority task, will get on the CPU. The amount of execution time that each task gets on the CPU is not enforced using a simple, reliable mechanism (like a hardware timer). Preemption according to priorities determines order of task execution, but it does nothing to control the duration that each task executes.

Question #2 (Software in Embedded Systems, cont'd)

(b) Assuming that a variety of hardware interrupts (not just timer interrupts) are present, rank the three architectures (from best to worst) with respect how well they can produce the fastest and most deterministic response to externally initiated events. Assume that all interrupt service routines (ISRs) are short and that most of the work required to respond to external events is handled by task software and not the ISRs. Carefully justify your answer and state any reasonable assumptions that you are making to determine your ranking.

[5 marks] (3) Pre-emptive multitasking, with appropriately chosen task priorities, should produce the fastest and most deterministic response to events. However, the priorities must be chosen so that the most important events are handled with the tasks with the highest priorities. Both the response times and the determinism get worse with lower and lower task priorities. At least the priority levels can be selected to give the best response performance to those events that need it.

[5 marks] (2) Periodically scheduled state-driven code can be structured so that tasks that must have faster and more deterministic response times can be assigned to lists that provide more frequent scheduling. The maximum response time for an event will be determined by the scheduled period of the corresponding task as well as the code in the implementation of the task states. Multiple events of equal importance can be placed on the same list with the same period, and so they could have roughly the same response times and determinism. All this presupposes that there is sufficient idle time so that the timing produced by the scheduling lists is accurate.

[5 marks] (1) Round-robin time slicing will produce the worst response time behavior with respect to speed and determinism. Time slicing assigns execution time to the tasks in a fair way, but this ignores the possibility that different tasks may be handling events with different response time requirements. Time slicing can be modified to produce faster response, say by reducing the duration of the slices, but that would increase the amount of time wasted doing more frequent context switches. Allocating more time slices to the more demanding tasks produces a system that starts to look like periodically scheduled state driven code.

Question #3 (MicroC/OS Implementation)

(a) In a MicroC/OS implementation, what is the purpose of the UserMain task? Which objects (e.g., variables, semaphores, message queues) should be instantiated outside of UserMain, and which objects should be instantiated inside UserMain? Where is the best place to initialize those objects? Carefully justify your answers.

[5 marks] UserMain is used to initialize system functions (e.g. TCP/IP stack) and shared data structure. It is also used to create all other user application tasks.

[5 marks] Shared "global" data structures, like variables, semaphores, message queues and stack space for the application tasks, must be instantiated outside the UserMain task. They are global in the sense that all application tasks can access them. (Note that in the C environment, these data structures are actually local to the main() program.) Other tasks would see a data structure that was instantiated inside one task. Objects that are only accessed by UserMain (e.g., private counter variables) should be instantiated inside UserMain.

[5 marks] UserMain is the appropriate place to initialize all global objects and also objects that are local to UserMain. However, constant objects, such as string labels, must be defined where they are declared outside UserMain. Global pointers are also best given initial values when created.

(b) What is the main purpose of the idle task in the MicroC/OS environment? What computations or other actions could be usefully done inside the idle task?

[5 marks] The idle task is used to safely consume CPU execution time that is not required by other software (e.g., threads, tasks, the operating system, exception handling routines). The idle task is not required to do any time-sensitive work (it may just execute no-operation instructions). It is never blocked and so is always available to absorb available unused CPU time.

[5 marks] The idle task could be used to increment a counter whose count would then be a measure of how much time was spent in the idle task. This time measure could then be used to calculate the fraction of time that is being used by the other "useful" software, which would be a measure of how busy the CPU was being kept. Other long-running calculations (e.g. back-ups, software updates, background system diagnostics, cryptocurrency mining) could also be performed by the idle task.

Question #4 (Communication Interfaces)

(a) In a receiver interface, what is meant by an "overrun error"? Given a hardware receiver design that produces receiver interrupts, what can be done to ensure that overrun errors do not occur in an embedded system.

[6 marks] An overrun error is the situation in a receiver (which has limited data storage capacity) when data is lost because new data arrives before the previously received data has been read out of the receiver's buffer by the CPU and relocated elsewhere. Thus either the previously stored data, or the newly received data, is lost due to the limited storage capacity in the receiver.

[6 marks] Overrun errors can be avoided if the receiver produces an interrupt when a new block of data (e.g., one character) is received, and the CPU responds promptly enough and reads the data out of the receiver's buffer and writes the data elsewhere to be processed. In some receivers (e.g., the UART in Freescale microcontrollers), an interrupt can be configured to fire when the receiver FIFO buffer is full. In the case of FIFO receiver interrupts, the entire contents of the FIFO buffer could be read out and relocated somewhere else.

(b) The Fast Ethernet Controller (FEC) block in the MCF5234 microcontroller uses two independent systems of buffers in the CPU's memory for storing the contents of Ethernet frames in the transmit and receive directions, respectively. The buffers in the transmit direction are variable in size, but the buffers in the receiver direction all have the same size. What are likely to be the major advantages and disadvantages of having to use buffers of the same size in the receive direction?

[8 marks] Advantages of equal-sized receive buffers:
(1) It is simpler to create a buffer management system when the buffers are all of the same size.
(2) It is not necessary to analyze the received data to decide how to partition it and save it into buffers of different sizes.

[5 marks] Disadvantages of equal-sized receive buffers:
(1) If a way could be found to partition the received data in different subblocks (e.g., different protocol headers and the payload), then equal-sized buffers might be wasteful since they many of them would be partly filled.
(2) (Related to (1)) It would be inefficient to distribute partly filled buffers to different software modules.

Note: Typically there is no easy way of partitioning the received data (as it is being received) into different subblocks, which could then be processed separately by different software layers (e.g. protocol layers).