UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Midterm Examination

Instructor: B. F. Cockburn
Exam date: February 27, 2019
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, Solutions
Family name **Given name**

Signature: _____

Student I.D.: _____

Question	Time	Worth	Mark	Subject
1.	10	20		Basic Concepts
2.	15	30		Software Architecture
3.	12	25		Critical Sections
4.	13	25		Communication Interfaces
Total	**50 mins**	**100**		**—**

Question #1 (Basic Concepts)

Carefully explain the following pairs of concepts, emphasizing the similarities and differences for each pair:

(a) The reference and dereference operators in C

[6 mark] The **reference operator &** (e.g., &objectInstance) obtains the base memory address of a software object. The address is a pointer value that should only be passed, unchanged, to functions or pointer operators (e.g., * and both + and – with an integer as the second argument). The **dereference operator *** takes a pointer (i.e., a variable containing a pointer value, which should contain the base address of a defined software object) and returns a name for the software object. The name (e.g., *PtrName) can be used on the left hand side of an assignment statement to update the value of the object; it can also appear in an expression that produces a value that is assigned to a variable or passed as an input parameter to a function or operator. The reference and dereference operators are inverses of each other.

(b) Automatic and dynamic memory allocation

[7 marks] **Automatic memory allocation** is the action where memory space is automatically allocated from the stack for local variables when execution of compiled code enters a new context (e.g., a function or a context defined by a pair of braces). That memory space is automatically recycled back to the stack (in a pop operation) when the context is exited. **Dynamic memory allocation** is when a block of memory space from the heap is requested explicitly (e.g., by calling the function malloc()) from the run-time environment to implement a variable or another object. The block of memory space is identified using a pointer to the base address of the block. The block must eventually be recycled explicitly by passing the pointer value to the free() function; otherwise, there is a memory leak and the block of memory will effectively become inaccessible.

(c) Cohesion and information hiding in module design

[7 marks]
Cohesion is the property where a software module (e.g. a function) is concerned only with closely related data and functions.

Information hiding is the property where a software module hides implementation details (e.g. private variables, data structures and functions) that do not need to be disclosed to users of the module. Thus users will not depend on the details.

The two properties are both desirable as part of good software design. However, they do refer to different qualities.

Question #2 (Software Architecture)

(a) It is common to use an embedded system to initiate events on a scheduled basis. For example, an embedded system might be used to repeatedly take measurements of critical parameters in an industrial plant. The timing of such internally initiated events is often specified in terms of a frequency (with a maximum allowed error with respect to the nominal frequency) and jitter (the worst-case short-term variability in the times when the events are initiated). What software architecture would you recommend for an embedded system that must (1) trigger temperature measurements at three sensors at a rate of 900 Hz, (2) trigger pressure measurements at two sensors at a rate of 300 Hz, and (3) trigger a flow rate measurement at a rate of 200 Hz. Your architecture is to be chosen to produce the greatest possible frequency accuracy and the least possible jitter for the measurement events. Carefully justify your recommendation in the space below.

[15 marks]
There are two basic software architectures that would be good choices for this problem: (1) multiple nonpreemptive loops, and (2) periodically scheduled state-driven code. In (1) a single thread would run "on bare metal", with a sufficiently fast repetition frequency, with iterations identified using a state variable. The three measurements would be made using functions that are called when the state variable had the appropriate values. In (2) the measurements would be made by three different tasks that have scheduling periods at 900 Hz, 300 Hz and 200 Hz. An operating system would schedule the tasks.

In both architectures one would much prefer to have harmonically related repetition periods so that the events do not overlap or interfere. The 900 Hz loop has a period of 1/900 sec, and the 300 Hz loop has a period of 3/900 sec: these periods are indeed harmonically related (3/900 is exactly three times 1/900). However, the period for 200 Hz is not harmonic with respect to the other two. Note that the period of 200 Hz is 4.5/900 sec. One simple solution would be to do the flow rate measurements at 300 Hz and then use an interpolation calculation to convert those measurements to 200 Hz.

Alternatively, one could set up (using a hardware timer) a fast timing loop that has a period T = 0.25/900 sec. The 900 Hz, 300 Hz and 200 Hz loops would then have periods of 4T, 12T and 18T, respectively. The ISR for the fast loop would maintain a state variable that increments a state variable from 0, 1, ..., 35 and then wraps around to 0. The flow rate measurement could be made at 900 Hz in states 2, 6, 10, 14, 18, 22, 26, 30, 34, 2, repeat. The pressure measurements could be made at 300 Hz in states 0, 12, 24, 0, repeat. Finally, the 900 Hz temperature measurements could be made in states 1, 19, 1, 19, 1, repeat. Note that in any one state, only one of the three kinds of measurements would made. This interleaving scheme would likely ensure an accurate repetition frequency and the least jitter.

Question #2 (Software Architecture, cont'd)

(b) Briefly describe the similarities and differences between the round robin time slicing and the preemptive multitasking software architectures. What are the major characteristics and advantages that are provided by each of the two architectures? Briefly explain why real-time event handling is best managed within the scope of a preemptive multitasking architecture as opposed to a round robin architecture. What would be an appropriate use of the round robin time slicing architecture?

[7 marks]
Round robin time slicing and preemptive multitasking are both software architectures that partition the application software into tasks that are stopped and restarted. In round robin, the tasks are ordered in a fixed list and each task in turn is given a fixed duration of execution time on the CPU. The context switches between running tasks are triggered by a stream of hardware timer interrupts. In preemptive multitasking, each task is given a priority and the highest priority ready-to-run task executes on the CPU. Each task (except the lowest priority task) must periodically encounter a blocking condition so that the running task is stopped from executing and the other tasks can have a chance to execute.

[4 marks]
Round robin time slicing ensures fair sharing of the CPU time among the tasks. Each task has a guaranteed fraction of the CPU time. It is possible to give some tasks more execution time by increasing their proportion of the available time slices per second. However, fair sharing of the CPU's time does not ensure that real-time events will be handled quickly and deterministically. The jitter could be quite large.

Preemptive multitasking uses a simple priority scheme to ensure that the CPU time is allocated first to some tasks (those with higher priorities) over other tasks (those with lower priorities). This means that better real-time performance (faster response, more predictable response) can be provided to handle the more time-critical events. However, the tasks must be designed so that all tasks (except the lowest priority idle task) will always eventually encounter blocking conditions that force them to give up the CPU. This, together with light loading, ensures that all tasks get execution time on the CPU.

[4 marks]
Round robin time slicing is ideal for sharing a large computer among a pool of users who do not require real time response.

Preemptive multitasking is best for implementing embedded systems that must provide real-time response for a number of different events that have different priorities and different real-time response requirements.

Question #3 (Critical Sections)

(a) Briefly explain what is meant by a "critical section" in a multitasking system? What can happen if steps are not taken to handle a critical section properly?

[7 marks]
A critical section is a section of code (e.g., data structure updates, hardware register updates) that, once started, must be executed until completion without task interleaving.

[5 marks]
If a critical section is not protected properly (e.g., by locking out task context switches) data structures and/or hardware registers may be left with inconsistent values. Inconsistent values may cause tasks to perform erroneous actions, quite possibly leading to serious system failure.

(b) Critical sections can be handled in a variety of different techniques, including using semaphores, temporarily locking out interrupts, and temporarily disabling multitasking. What the major advantages and disadvantages of these three techniques with respect to each other?

[5 marks]
Semaphores can be used to control access for a pool of tasks to one critical section. All tasks that could access the critical section must be in that pool. A semaphore, which is initialized to 1, is used to ensure that only one task enters the critical section at a time. A task must pend on the semaphore before entering the critical section, and must post on the semaphore when leaving the section. When a task in the critical section, the semaphore has the value 0, which blocks any other tasks from entering the critical section. Adv. The method only affects the tasks that use the critical section. Disad. Need semaphore data structure and pend() and post().

[4 marks]
Temporarily locking out interrupts at the start of a critical section is a fairly drastic but effective method for protecting a critical section. Adv: It is simple, and all other tasks will be prevented from executing. Disadv: It disturbs the execution of all tasks, even those that don't use the critical section.

[4 marks]
Temporarily disabling multitasking protects the critical section in a simple way without disabling interrupts. Adv. Simple method. Interrupt-driven functions can still proceed. Disadv. It must be certain that the execution of interrupt service routines cannot corrupt the critical section.

Question #4 (Communication Interfaces)

(a) What is meant by an "underrun error" in a transmitter as opposed to an "overrun error" in a receiver? What features can be provided in the transmitter hardware and associated software to ensure that underrun errors are less likely to happen?

[12 marks]
An underrun error in in a transmitter is the situation when the transmission of a data packet has started, but the source of data (e.g., the CPU or a DMA controller) failed to keep up. As a result the transmission of the data packet stopped early and was abandoned and the underrun error condition became true.

An overrun error in a receiver is the situation when recently received data was not read out fast enough (usually by the CPU) and so newly arrived data was overwritten and lost.

The simplest form of protection against underrun and overrun errors is to have sufficiently large first-in first-out (FIFO) data buffers in the transmitter and receiver, respectively. Also, an interrupt could be generated by the hardware whenever the transmitter buffer is getting too empty, or whenever the receiver buffer is getting too full.

(b) The Ethernet standards use line codes like the Manchester code and the 4B5B line code. Briefly explain what those line codes are. (You do not need to write down the 4B5B line code table in your answer.) What is the main benefit of using these line codes?

[5 marks]
The Manchester code replaces each 0 data bit with a low-to-high signal and each 1 data bit with a high-to-low signal. (The code can also be defined with the signal directions reversed.) In this way an encoded sequence of data bits contains at least one signal transition per bit.

[5 marks]
A 4B5B line code replaces each block of 4 data bits with a 5-bit codeword that contains at least one signal transition. Each of the 16 4-bit data blocks is associated with a different 5-bit codeword.

[3 marks]
The Manchester code and 4B5B line code are both used to ensure that there will always be a sufficient density of transitions (L to H, or H to L) in the data signal so that the clock signal can be recovered reliably at the receiver from the encoded data signal. The Manchester code effectively doubles the bit rate of the data signal whereas the 4B5B code increases the bit rate by 25%.