UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Midterm Examination

Instructor:	B. F. Cockburn
Exam date:	February 26, 2020
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. One two-sided 8.5" × 11" formula sheet of any design can be consulted. No other references, including model solutions, can be consulted. The Internet cannot be accessed using 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.	18	35		Software Architecture
3.	12	25		MicroC/OS
4.	10	20		Interrupt-driven Communication
Total	**50 mins**	**100**		—

Question #1 (Basic Concepts)

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

(a) Blocked task vs. interrupted task [7 marks]

A **blocked task** is a task whose execution cannot proceed because some condition (e.g., a semaphore is 0, or a message queue is empty, or a requested time delay has not yet elapsed) is not yet true. A blocked task cannot be interrupted because it is not executing; only an executing task can be interrupted.

An **interrupted task** is a task that was temporarily stopped from executing because the interrupt handling mechanism of the CPU decided to respond to a hardware interrupt signal by suspending execution of the interrupted task and beginning execution of the appropriate interrupt service routine. An interrupted task is not blocked: its execution is not being held back by a condition; rather, the interrupt mechanism temporarily caused the task to stop executing to allow an interrupt service routine to run and handle some interrupt-causing event.

(b) Dangling pointer vs. NULL pointer [7 marks]

A **dangling pointer** is a pointer value (a reference) to a block of memory that was dynamically allocated at one time but has since been returned to the heap. A dangling pointer should not be used since the memory that it is referencing may be either out-of-service in the heap or dynamically re-allocated for another purpose. The value of a dangling pointer (unlike NULL) does not reveal, by itself, that it is not a valid reference.

A **NULL pointer** is a constant value (binary 0 by convention in C) that is used to indicate an uninitialized pointer value. A pointer with value NULL must not be dereferenced since it does not contain a valid reference to an instantiated object. A NULL pointer is not dangling since the value NULL was never previous used to reference an allocated object.

(c) Software modules with high cohesion vs. loosely coupled software modules [6 marks]

A software module with **high cohesion** gathers together only data structures and functions that are closely related. A module with high cohesion will tend to be more loosely coupled with other modules.

Two or more software modules that are **loosely coupled** have relatively few interactions. They will tend not to share data structures. They will interact through relatively few function calls and/or synchronization mechanisms (e.g. semaphores, message queues). They will tend to have high cohesion.

Question #2 (Software Architecture)

(a) Some embedded systems are most easily implemented using a single, non-preemptive thread that is running "on bare metal", without the presence of an operating system. The single thread in such a system begins to execute a main loop when enabled by an interrupt that is produced by a hardware timer. How can such a software architecture be designed to produce polling operations that loop at two or more different repetition frequencies? Briefly explain why the periods of all of the different polling loops should be harmonic. What does "harmonic periods" mean in this situation?

[15 marks]
To produce the required timing, a **hardware timer** should be used to produce interrupts at a suitable faster frequency. The main nonpreemptive software loop busy waits on a flag that is set to 1 by the timer interrupt service routine (ISR). Once the flag is set, the main loop proceeds, clears the flag to 0 and executes polling code that is selected according to the state of a state counter. The state counter is incremented after each interrupt, and wraps around back to zero after a maximum count is reached.

The periods in a set of two or more periods are said to be **harmonic** if each period is a whole multiple of all of the smaller periods than itself. Harmonic periods are desirable because it is relatively easy to interleave the period starts so that they will never coincide in time with each other, or shift by each other in time.

All of the different loop periods must be whole multiples of the accurately generated repetition period T_timer of the hardware timer. For example, if the three harmonic loop periods are T_1, T_2 and T_3, then T_1 = N_1 x T_timer, T_2 = N_2 x T_timer, and T_3 = N_3 x T_timer, for three integers N_1, N_2 and N_3.

(b) In multitasking software architectures the application software is implemented as one or more tasks, which must share execution time on the Central Processing Unit (CPU). There are a variety of different architectures that determine how the tasks will share the CPU including: (1) round-robin timing slicing, (2) periodically scheduled state-driven code, (3) cooperative multitasking, and (4) preemptive priority-based multitasking. For each of these four architectures very briefly give the major advantages and disadvantages. Why is architecture (4) generally preferred among the four choices when implementing embedded real-time systems?

[20 marks]

Round-robin time slicing has the advantage of timer-enforced sharing of the CPU's time among the tasks. The CPU time allocated to each task is directly controlled and does not have to be equal. The disadvantage is that the time sharing is rather rigid, will not respond to workload changes, and does not provide fast or deterministic real-time response.

Periodically schedule state-driven code has the advantage of providing direct control over the frequency with which the different states of state-organized tasks are executed. This is architecture convenient when the tasks implement finite state machines. The architecture does not provide very deterministic response to externally generated real-time events.

Cooperative multitasking has the advantage of minimizing the number of task-to-task context switches, and this minimizes the time spent on those switches. However, this architecture has poor real-time response to events. Interrupts can be used to generate timer-controlled signals and external event handling, but the longer processing of information by tasks will not have fast or deterministic real-time response.

Preemptive priority-based multitasking associates priorities with tasks, and always gives the CPU to the highest-priority ready-to-run task. This gives the designer control over the quality of real-time service, but it can make detailed system behaviour complex and hard-to-debug. Problems, like deadlock and priority-inversion, must be considered and handled.

Pre-emptive priority-based multitasking is generally preferred when implementing embedded real-time systems because by choosing the task priorities, the system designer has fairly direct control over the quality of the real-time response for the different tasks. The quality of service for a task can be adjusted later simply by changing the task's priority. New tasks can be introduced into an existing system, and the priority of the new task can be selected so that the real-time performance of the new system will continue to meet the system's real-time specifications.

Question #3 (MicroC/OS)

(a) In a MicroC/OS application source file, where are the appropriate places in the C code to instantiate a semaphore, initialize a semaphore, post to a semaphore, and pend on a semaphore. Briefly justify your answer for each of the four sub-questions.

[13 marks]
A semaphore must be **instantiated** in the main program, outside all of the tasks, so that it will be accessible to any task.

A semaphore should be **initialized** inside of the UserMain task, which is the preferred place to do initializations since this task starts executing before any other user tasks begin to run.

Semaphores are **posted** to by either tasks or interrupt service routines (ISRs). If one or more tasks are blocked on a semaphore, then the post operation unblocks the highest priority blocked task. An ISR can safely post to a semaphore since the post operation will not block the ISR. (ISR execution cannot be blocked.)

Semaphores are **pended** to by tasks. A task that blocks on a semaphore will get blocked if the decremented value of the semaphore's count is negative.

(b) Briefly describe the situations when a call to OSSemPend() will cause a context switch. Repeat the question for a call to OSSemPost(). Be sure to clearly indicate what conditions must be true for the context switch to occur in each case.

[12 marks]
A call by a task to OSSemPend() for a semaphore will cause a context switch if the semaphore count, when decremented, has a negative value. (Equivalently, if the initial semaphore count is less than or equal to zero, then a context switch will occur when OSSemPend is called.) The calling task is blocked and a context switch occurs so that the corresponding protected resource (e.g., a critical section) remains protected from the calling task.

A call by a task to OSSemPost() for a semaphore will cause a context switch if the OSSemPost() unblocks another task that has a higher priority than the currently executing task. The semaphore count must start out with a value of at least zero, and there must be at least one other blocked task that has a greater priority than the currently running task.

Question #4 (Interrupt-driven Communications)

(a) Interrupt-driven data transfer is an efficient and widely used method for sending data out through a transmitter interface, or receiving data in through a receiver interface. What makes this method so efficient compared to the alternative of busy waiting?

[8 marks]
Interrupt-driven data transfer is efficient because the CPU is not required to spend time busy waiting on a status register for the communications interface. The CPU only has to spend time on the data transfer after the hardware has alerted it (through an interrupt) that there is useful work to be done. Thus the CPU is freed up to do other useful work.

(b) In interrupt-driven communications, which software is responsible for enabling interrupts to start a data transfer, and which software is responsible for disabling the interrupts at the end of the data transfer? What would happen in each case if the enabling or disabling software code were to be absent?

[12 marks]
Interrupts in the receive direction are enabled when data is expected to arrive. If the receive interrupts are not enabled, then received data will get lost if there is insufficient memory storage space in the receiver hardware to hold it.

Interrupts in the transmit direction are enabled only after the system has data to send. The interrupt fires as soon as there is room in the transmitter to accept more data to be sent. If transmit interrupts are not enabled, then the transmission will never start.

Interrupts in the receive direction are disabled as a measure to prevent the further reception of data. This would only be done if it is certain that no more useful data can be received because any such arriving data will run the risk of being lost if it exceeds the storage capacity of the receiver hardware. If receive interrupts are not disabled, then possibly unwanted data will be received and possibly processed in error.

Interrupts in the transmit direction are disabled when there is no more data to be sent. If the interrupts are not disabled, then the interrupt service routine (ISR) will have no data to load into the transmitter hardware. If the ISR returns, then another interrupt will immediately happen and will face the same problem: there is no data to transmit. A rapid series of pointless transmit interrupts will likely result.