UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Midterm Examination

Instructor: B. F. Cockburn
Exam date: October 24, 2012
Exam duration: 50 minutes
Aids permitted: A paper or electronic copy of the course lecture slides can be freely consulted.
The Internet cannot be accessed by any device.
Electronic calculators of any kind are permitted.
The model solutions for this year's Assignments #1 and 2 can be consulted.

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.	8	15		Basic Concepts
2.	12	25		Software Architecture
3.	18	35		MicroC/OS
4.	12	25		Communications Interfaces
Total	**50 mins**	**100**		—

Question #1 (Basic Concepts)

Summarize the similarities and differences of the following pairs of concepts:

(a) Microcontroller vs. digital signal processor

[5 marks] A microcontroller contains a microprocessor (i.e., central processing unit) plus a variety of supporting peripheral systems (e.g., communications ports) on the same integrated circuit. A digital signal processor (DSP) is a microprocessor that has special features (e.g., instructions, parallel data paths) that allow it to be especially efficient at performing numerically-demanding calculations. Microcontrollers are intended for implementing compact embedded systems, where as many functions as possible are packed into one integrated circuit. A DSP is a specialized microprocessor that aims to provide great numerical performance. A DSP won't usually be intended for embedded control applications on its own. This being said, as the number of available transistors per chip continues to increase, the microprocessor(s) on micro-controllers will quite likely include typical DSP features (e.g., high-spead multiplier-accumulator hardware and instructions).

(b) RISC vs. CISC

[5 marks] A complex instruction set computer (CISC) is a computer that has a relatively large instruction set with a rich set of addressing modes. CISCs are intended to support assembly language programming as well as programming using compiled high-level languages. A reduced instruction set computer (RISC) is one that has a reduced number of instructions (compared to CISCs) and a reduced number of addressing modes. The reductions in a RISC are intended to simplify hardware implementation (e.g., using simpler and deeper pipelines) and to produce higher computational speed (despite the less powerful individual instructions). The vast bulk of code produced for RISCs will be compiled code, and the simplified RISC architecture should help make the compiler's job easier.

(c) Soft real time vs. hard real time

[5 marks] "Hard real time" refers to computer applications where there are relatively demanding and inflexible maximum response time constraints to externally-triggered events. If hard real time constraints are not met, then serious system failure may result. A special operating system or kernel must be used in hard real time applications. "Soft real time" refers to applications where the computer must respond reasonably quickly to external events. However, the implications of missing response time deadlines are not a serious as for hard real time applications. Soft real time systems can often use lightly loaded conventional operating systems, such as Windows or Unix.

Question #2 (Software Architecture)

(a) In this course, we use MicroC/OS, which implements pre-emptive multitasking with at most one task for each priority level. The highest priority task that is ready-to-run gets to run on the CPU. Once the highest priority ready-to-run task is running on the CPU, what constraints must be respected in the software design to ensure that tasks at other priorities will also get a chance to execute on the CPU.

[8 marks] Every task (except possibly the lowest-priority application task) must block its own execution periodically (e.g., semaphore, message queue, timer delay) to allow lower priority tasks a chance to execute on the CPU. Preemptive multitasking systems do not directly ensure fairness in the sharing of CPU time among tasks, so the tasks themselves must voluntarily adopt constraints (e.g. use a looping structure with blocking event(s) in the loop) to allow fair sharing of the CPU. Hardware interrupts can also be used to unblock tasks, say by posting to a semaphore or by appending a message to a message queue, but interrupts are not a complete solution to the CPU sharing problem. A task can also lower its own priority to cause context switches, but this is approach would be relatively complex and would essentially produce a form of cooperative multitasking.

(b) Briefly explain how the occurrence of a hardware interrupt can cause a context switch in a pre-emptive multitasking kernel. How could the running task prevent any context switches, even those caused by interrupts? What would be the danger in doing this?

[8 marks] A hardware interrupt is serviced by the execution of an interrupt service routine (ISR), which is called an exception handling routine (ESR) by Freescale Semiconductor. Execution of the ISR/ESR can involve posting to a semaphore or adding a message to a message queue, which are events that can unblock a higher-priority task and thereby make it ready-to-run. If the priority of an unblocked task is higher than that of the interrupted task, then the kernel will not just restart the interrupted task; rather, it will do a context switch and restart the highest-priority ready-to-run task instead.

[6 marks] A running task could prevent context switches, while permitting interrupts, by calling the routine OSLock(). Later on, the running task must restore multitasking by calling OSUnlock(). A more disruptive approach would be to disable all maskable interrupts by calling the routine USER_ENTER_CRITICAL(). With interrupts disabled, the unwanted task context switches would be avoided. The task must later on call USER_EXIT_CRITICAL() to restore interrupt processing.

[3 marks] Disabling or masking out interrupts is undesirable since it will delay the processing of other possibly important events. The determinism of event handling would be less predictable.

Question #3 (MicroC/OS)

(a) As noted above, MicroC/OS was designed to support pre-emptive multitasking, with at most one task at each of 63 possible priority levels. However, MicroC can also be used to implement other forms of multitasking. In cooperative multi-tasking, once a task starts to run it can run as long as it wants before giving up the CPU to the next task in a ring of tasks. Briefly explain how you could use binary semaphores in MicroC/OS to implement cooperative multitasking. Be sure to explain how each task would be allowed to start executing, and how it would give up the CPU to the next task. Also explain where in the code and to what values the semaphores would be initialized. Hint: In your design provide a low-priority referee task that posts to semaphores.

[20 marks]

Let's take the hint and include a low-priority task that posts to semaphores. We will use a separate semaphore to enable each task. So if we have tasks A, C and C, those tasks can be enabled and disabled by semaphores SemA, SemB and SemC.

For the execution of each task to be controlled effectively by one semaphore, the task must be structured into a loop so that it periodically "pends" on the semaphore to give up the CPU. Also, the task should immediately start with a "pend" to the semaphore. We will ensure that at all pend times (except possibly the first), the reference task is the only ready-to-run task, and so a context switch will occur back to the referee task. The execution time between subsequent pends should be close to the correct desired execution time for each time slot.

The semaphores will all be created and initialized to zero, and the referee task will start to run before all other user tasks. All of the non-referee tasks will thus start out blocked on their respective semaphores. The referee task maintains a scheduling table with one task in each row. The tasks can be given different time slot intervals, or the same duration of interval. The referee task then passes down along the rows of the scheduling table, and enables each task in turn by "posting" to the corresponding semaphore. Once the last row in the table is reached, the referee task wraps around back to the first row.

One can implement a solution that does not use a referee task, by requiring each task to post to the semaphore of the next task. But this solution would be less desirable since the control of task execution would be distributed across multiple tasks instead of being gathered together in one place (inside the referee task).

Question #3 (MicroC/OS, cont'd)

(b) The input and output parameters of the functions in MicroC/OS are almost all defined to have abstract symbolic types, like BYTE, WORD, OS_SEM or OS_Q. These types are then given detailed, but hidden, definitions in C inside MicroC/OS header files. For some other parameters, the special C types "void" or "void *" are used. Briefly explain why symbolic types were used in some cases, while "void" types were used in other cases. What is the disadvantage to using "void" types?

[5 marks] The symbolic type definitions are used in cases where the type of the parameter is fixed, but it is preferred to keep the implementation details hidden (to obtain the usual benefits of information hiding). For example, by hiding the implementation details, it is easier to alter the operating system without having to modify the source code of the applications. Thus using symbolic types increases portability. Only a recompilation would be necessary to move to different CPU hardware. Also, symbolic types are more readable and self-documenting.

[5 marks] The "void" and "void *" types in C have very different semantics (i.e., meaning). A "void" input parameter to a function means "no input parameters are present". Similarly, a "void" output parameter to a function means "no output parameter is present".

In cases where it would be convenient to have parameters of different types, or in cases where the parameter types are not known ahead of time, then the "void *" type may be used as an intermediate typeless pointer for passing parameters into and out of functions. If pointers of type "void *" were to be dereferenced, the result would be a variable of type "void", which can't be used because there are no constants or variables of type "void". Instead, the type casting feature of C is used to convert typed pointers (e.g., "int *") to and from typeless pointers of type "void *". To pass a typed pointer into a function as an input parameter, one first converts it to a typeless "void *" pointer. To use a typeless "void *" output pointer resulting from a function, the pointer can first be cast to a typed pointer and then dereferenced to a typed constant or variable.

[5 marks] The disadvantage of using "void *" types is that the programmer is effectively turning off type-checking. Run-time type errors can then occur without warning, and serious system failures could result. For example, after being cast to a "void *", there is no type difference between a pointer to a 32-bit integer and a pointer to a 64-bit integer; but casting such a pointer to a typed pointer and then dereferencing it would produce different results. The pointer types, before and after being cast to a "void *", should be the same. Casting to the wrong typed pointer type would likely produce serious run-time errors.

Question #4 (Communications Interfaces)

(a) We saw in the lectures that both the UARTs and the FEC in the MCF5234 can temporarily store multiple bytes in both the transmit and receive directions. What are the most important reasons for providing temporary storage in this way?

[8 marks] Providing temporary storage (in both the Tx and Rx directions) is convenient because it decouples the systems on either side of the interface, in both directions. Small short-term mismatches in data rates can be safely accommodated in the FIFO. The design of the systems on either side of the interface is simplified because the requirement for matching the data rates is relaxed. Also, the number of interrupts (and hence the interrupt processing overhead) can be reduced if interrupts can be triggered over the state of the entire buffer (e.g., Rx buffer full or Tx buffer empty) instead of on a character-by-character basis.

(b) In the UART, why are there four bytes of temporary storage in the receive direction, but only two bytes of storage in the transmit direction?

[8 marks] A larger storage capacity (4 vs. 2 bytes) is used in the Rx direction vs. the Tx direction to give the CPU extra safety margin in responding to the arrival of characters in the direction (receiver) over which it does not have direct control.

(c) In the FEC, the storage capacity of the first-in first-out (FIFO) buffer is split in a programmable way between the transmit and receive directions. If one were to use the FEC in the faster 100-Mbps mode (Fast Ethernet instead of the original 10-Mbps Ethernet), what would be important advantages to allocating more space in the FIFO to the transmit direction as opposed to the receive direction?

[9 marks] With Fast Ethernet, one might wish to reduce the risk of underflow (the Tx FIFO going empty during transmission) by allocating more room in the shared FIFO to the Tx direction. However, this would increase the risk of FIFO overflow in the Rx direction. One should consider the expected Tx and Rx data rates, as well as the likely response time of the CPU, in order to intelligently partition the FIFO between Tx and Rx. For example, if a server produces far more data than it receives, then the Tx FIFO partition might be made larger than the Rx FIFO partition.