UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CMPE 401 – Computer Interfacing

Final Examination

Instructor: B. F. Cockburn
Exam date: December 19, 2011
Exam duration: 120 minutes
Aids permitted: A hardcopy of the course lecture slides can be freely consulted.
Model solutions for the Fall 2011 assignments and midterm can be consulted.
No other model solutions are to be consulted.
One two-sided 8.5" x 11" formula sheet can be consulted.
Electronic calculators (all kinds) are permitted.

Instructions: 1. Enter your printed name, signature and I.D. number on this cover page.
2. Verify that this booklet contains 10 pages (including this cover page).
3. Neatly enter your answers in the spaces provided.
4. Use the reverse sides of the pages for extra space or rough work.

Student name: **_____Model_____,____Solutions_____**
Last name **First name**

Signature: **_____**

Student I.D.: **_____**

Question	Time	Worth	Mark	Subject
1.	15	12		Fundamental Concepts
2.	20	17		Multitasking Systems
3.	20	17		MCF5234 Peripherals
4.	15	12		Semaphores in MicroC/OS
5.	30	25		Performance Trade-offs in TCP/IP
6.	20	17		Time Sharing in the eTPU
Total	**120 mins**	**100**		---

Question #1 (Fundamental Concepts)

In your own words briefly define each of the following concepts. Be sure to explain why each concept is important in Computer Interfacing.

(a) Deadlock

[2 marks] Deadlock is an undesirable situation in a multitasking system where a set of tasks are blocked because they meet the following four conditions: (1) each task has been given exclusive use of a resource, (2) each task requires at least two resources, (3) none of the tasks can be forced to release a resource, and (4) the tasks form a ring such that each task in the ring is waiting on at least one other resource that is exclusively held by another task in the ring.

[2 marks] Deadlock is a situation that must be avoided (or at least detected and fixed soon afterwards) since the affected tasks will be effectively prevented from getting unblocked and executing. Serious consequences might result from the occurrence of deadlock.

(b) Rate Monotonic Fixed Priorities

[2 marks] Rate monotonic (RM) fixed priorities is a strategy for assigning priorities to tasks within a pre-emptive multitasking system with fixed priorities. Tasks that must execute more frequently are assigned higher priorities.

[2 marks] Some reasonable strategy is required for assigning priorities in a pre-emptive multitasking system, ideally a strategy that maximizes the likelihood that all system functions are successfully completed in a timely manner. The RM strategy is useful for ensuring that frequently executing tasks don't fall behind in their duties. Such tasks should benefit from the RM strategy since they are probably more time-sensitive, and RM should reduce the scheduling time jitter.

(c) Integrated Development Environment

[2 marks] An Integrated Development Environment (IDE) is a single computer application that offers a consistent and comprehensive set of tools that can be used to develop software. The tools include code editors, source version control, compilers, linkers, debuggers, etc.

[2 marks] IDEs are intended to support software developers by improving their efficiency and increasing the likelihood that the produced software will be of high quality, with few software defects. The consistency of the tools (e.g. similar user interface) is intended to minimize mental mode switching and programmer fatigue, which lowers productivity and increases the chance of coding errors.

Question #2 (Multitasking Systems)

(a) It is desirable that a real-time kernel have a fast and predictable response time to external events. In fact in many applications, the predictability (also called determinism) of the response time is often viewed as being more important that the speed of the response time. What would be the major reason for preferring predictability in response over the average speed of the response.

[5 marks] It is difficult to compensate for unpredictability in the response time to external events. Unpredictability is a permanent loss in control over timing. Higher predictability, even if combined with increase in response time delay, would be preferable since the system can likely be designed to compensate for the increased (but predictable) response delay. Greater timing predictability also simplifies system modeling and debuggability.

(b) Why is it convenient have a lowest-priority "idle" task in a pre-emptive multitasking system? Briefly describe how an "idle" task could be modified to update a variable that gives the estimated load (e.g., percentage that the CPU is busy doing useful work).

[6 marks] An idle task is useful for (1) safely occupying the execution time that is not required by the other functional tasks, and (2) providing timing flexibility that can accommodate fluctuations in the timing demands from the other tasks. If the amount of idle time in a multitasking system were to be reduced to zero, then the system response time would rapidly increase to unacceptable delays. Using an idle task allows some useful low-priority work to be performed with the idle time.

[6 marks] The idle task could be easily modified to count timer ticks (or to just busy wait while incrementing a counter) and thus make a measurement of what percentage of the CPU's time was being consumed by the idle task. By substracting this idle percentage time from 100% of the time, one obtains the fraction of the CPU's time that is being consumed by the non-idle tasks. The calculation should be repeated on a periodic basis, where the period is ideally determined by interrupts from a hardware timer.

Question #3 (MCF5234 Peripherals)

(a) The three UART blocks in the MCF5234 microcontroller have multiple data registers in both the transmit and receive directions. Having multiple data registers provides flexibility that reduces the likelihood of data loss. Briefly explain how having multiple data registers also increases the efficiency of interrupt-driven input/output that uses the UARTs.

[7 marks]
Having multiple data registers in the transmit direction allows the device driver to load multiple pieces of data (e.g., bytes) in the same software action (e.g., interrupt service routine). Then the hardware can take care of the details of transmitting that data without further intervention from the CPU. In the receive direction, several pieces of data (e.g., bytes) can be recovered by the hardware without needing to trigger several software calls (e.g., multiple ISRs). Thus in both directions, the software overhead per data (and hence the software efficiency) is increased with multiple data registers.

(b) The Ethernet Interrupt Event Register (EIR) in the FEC, and the Channel Interrupt Status Register (CISR) in the eTPU, are controlled in similar ways. Briefly explain how, in general, bits are set and cleared in these two registers. You do not have to list and describe all of the possible interrupting conditions. In particular, why does writing a bit to 0 in either the EIR or CISR have no effect on the value of that bit? Why was this nonintuitive bit clearing mechanism adopted in the MCF5234 design?

[7 marks]
Bits are set in both of these registers as a result of the occurrence of specific hardware conditions. These hardware conditions can in term produce an active hardware interrupt signal if the corresponding bit in a mask register is also set.

Once set, the bit will be cleared if one of two things happen: (1) there is an active chip-level reset, or a reset that affects the entire interface block. (2) The bit is written by the software to 1, which nonintuitively causes the bit to be cleared to 0. The convention in (2) arises because an encoding was required that had one value that would leave the bit unchanged, and a second value that would clear the bit. The designers decided to redefine “write a 0 to the bit” to mean “leave the bit unchanged”, and “write a 1 to the bit” to mean “clear the bit to zero”. With this convention, a conventional bit mask can be created by OR-ing one or bit masks together, and then the resulting overall bit mask can be written to the status register to clear those bits in one operation. The conventional alternative would require (1) reading the status register, (2) clearing the desired bits, and (3) writing the new value to the status register.

Question #3 (MCF5234 Peripherals, cont'd)

(c) The Enhanced Time Processing Unit (eTPU) has a large number of memory-mapped registers. Some of these registers present the corresponding information (e.g., the Channel Interrupt Status Register) across all of the channels. Other registers present information on a per-channel basis (e.g., Channel *n* Status/Control Register). In fact, the same bits appear in the two kinds of registers. Why do you suppose this was done in the design of the CPU interface to the memory-mapped registers of the eTPU?

[3 marks]
The deliberate duplication of the same bits in different registers was done for the convenience of the software. Some software drivers (e.g., the eTPU-level interrupt service routine) will prefer to access and inspect all of the interrupt-causing bits from one register in one read operation. Other software drivers (e.g., channel function drivers) will prefer to see the bits associated with one channel gathered together in a small number of registers related only to that one channel.

Question #4 (Semaphores in MicroC/OS)

Consider the following three code fragments, which appear in UserMain task, a first user task A, and a second user task B:

```
// Before all tasks
OS_SEM MySemaphore;

// Fragment in UserMain
BYTE RetCode;
RetCode = OSSemInit( &MySemaphore, Value );

// Fragment in Task A
BYTE RetCode;
RetCode = OSSemPend( &MySemaphore, Value );

// Fragment in Task B
BYTE RetCode;
RetCode = OSSemPost( &MySemaphore );
```

(a) Briefly describe what happens when each of the two lines (after the comment line) in the UserMain fragment are executed. What determines the appropriate value to use for the parameter called "Value"?

The first line creates an instance of the object type OS_SEM that is called "MySemaphore". The constructor function is then automat-ically called to initialize MySemaphore.

The second line creates a variable of type BYTE that is called "RetCode". The variable is used to hold the subsequent return code.

The third line calls the MicroC function "OSSemInit()" and provides a reference to MySemaphore and the contents of variable "Value" as the two input arguments. The return code is loaded into variable "RetCode". Value passes the initial nonnegative count for the semaphore, which is normally 0 for a binary semaphore and the number of available resources for a counting semaphore. The function initializes a queue that records tasks that are blocked on the semaphore

(b) In the code fragment from Task A, briefly describe what happens when the third line of code (the call to OSSemPend) is executed. What value should be used for "Value"?

When OSSemPend() is called and the count for &MySemaphore is greater than zero, then the count is decremented by 1 and the calling task continues to execute the code following OSSemPend(). If the count is 0 or less, then the semaphore count is still decremented, and the calling task is blocked and placed on a queue of tasks that are blocked on semaphore &MySemaphore. A context switch is initiated to start execution of the next ready-to-run task.

The nonnegative value of "Value" specifies the maximum number of timer ticks for which Task A can be blocked on the semaphore. If the blocking time exceeds this time, then the function OSSemPend() returns with value OS_TIMEOUT. If "Value" has value 0 then Task A can be blocked on the semaphore for an indefinite time. If no timeout occurred, the function returns with value OS_NO_ERR.

Question #4 (Semaphores in MicroC/OS, cont'd)

(c) In the code fragment from Task B, briefly describe what happens when the third line of code (the call to OSSemPost) is executed. What conditions are indicated by the value of the return code?

When the function OSSemPost() is called, the scount associated with semaphore &MySemaphore is incremented by one. If there was at least one task blocked on the semaphore, then that task's record is removed from the semaphore queue and is added to the ready-to-run queue. If the unblocked task has a higher priority than the currently running task then a context switch is triggered.

The return code has value OS_SEM_OVF if the post operation causes the semaphore count to exceed the maximum allowed value; otherwise, the return code has value OS_NO_ERR.

Question #5 (Performance Trade-offs in TCP/IP)

The Transmission Control Protocol (TCP) uses a window-based flow control method, which simultaneously solves several important challenges that are encountered when operating a data connection over a potentially unreliable network. TCP provides a reliable bidirectional byte transportation service between a transmitting node and a receiving node.

(a) Briefly explain what the window is for one direction in a TCP connection.

The window is the number of data bytes that can be sent by the transmitter over one direction of the TCP connection without having been acknowledged (over the opposite direction link) by the receiver. The window is chosen by the receiver and sent back to the transmitter in a special field in packets that are returning (in the opposite direction) from the receiver to the transmitter.

(b) What would be appropriate considerations to take into account when deciding how big the window should be? What would happen if the window size were to be too small? On the other hand, what would happen if the window size were to be too big?

Having a larger window size gives more timing slack and hence flexibility to the communication link, but it also requires the receiver to be prepared to receive larger number of data bytes in a short period of time. If the link is inherently very reliable (e.g., an optical fibre connection) and high bit rate (with potentially many bits in transit over the link) then a large window size should lead to more efficient operation.

Having a smaller window size is a more cautious approach since it requires transmitted data bytes to be acknowledged more promptly. If the link is inherently unreliable (e.g., a relatively poor wireless link) then a smaller window size avoids wasteful repeated transmissions of data.

Having a zero window size allows the receiver to entirely prevent the transmitter from sending any data. Thus the setting the window size to 0 has the same effect as transmitting an X-OFF command in an RS-232 connection.

If the window size is too small, then a reliable and high-speed communication link will be poorly utilized. If the window size is too large, then a receiver with limited data buffer capacity could get overwhelmed with a sudden burst of data, leading to data loss.

Question #5 (Performance Trade-offs in TCP/IP, cont'd)

(c) TCP/IP datagrams can have variable length. The typical length of datagrams tends to be about 1500 bytes; however, there are situations where there might be advantages to having longer datagrams or shorter datagrams. First, briefly explain why TCP/IP datagrams tend to be 1500 bytes long. Second, briefly describe (and explain why) situations where longer datagrams would have advantages. Finally, briefly describe (and explain why) situations where shorter datagrams would have advantages.

TCP/IP datagrams tend to be about 1500 bytes long because of the similar packet length limit adopted by the Ethernet Local-Area Network (LAN) standard. Many computers are connected to the Internet through an Ethernet LAN, which thus acts to limit the length of the datagrams that are transmitted onto the Internet.

Longer datagrams are more efficient when a highly reliable communication medium (e.g., optical fibre) is being used. When the medium is very reliable, errors are rare and hence packet retransmissions are rare. Efficiency is increased by reducing the percentage of the bits that are overhead bits (e.g., header bits), and this reduction can be achieved by increasing the size of the payload field in each datagram.

Shorter datagrams are more efficient if the communication medium is unreliable (e.g., a noisy wireless channel) because the datagram header is used to recognize the occurrence of transmission errors. It is inefficient to tie up the communication medium with a datagram that contains errors, and using short datagrams allows corrupted datagrams to be detected sooner.

Question #6 (Time Sharing in the eTPU)

There are many similarities between the time sharing problem among multiple software tasks that is managed by the MicroC/OS kernel, and the time sharing problem among multiple hardware channel circuits that is managed by the scheduler in the Enhanced Time Processing Unit (eTPU).

(a) How does the eTPU scheduler ensure that all tasks, no matter what their priority, will get some execution time on the microengine? What prevents low-priority tasks in the MicroC/OS environment from being entirely prevented from getting any execution time in competition with higher-priority tasks?

The eTPU scheduler uses a seven-slot round-robin priority scheme in which high-priority tasks (called eTPU channels) are given top priority for four slots, and medium-priority tasks are given top priority in two slots, and low-priority tasks are given top priority in one slot. Within each priority each task is awarded execution time slots on a strictly round-robin basis. Thus all tasks, at all priorities, will eventually get some execution time (assuming that all time slots are finite in duration).

Low-priority tasks in MicroCOS can be entirely shut out from executing if the higher-priority tasks leave no time available. But such a situation would be disastrous since some idle time is always required to provide sufficient slack to keep all of the system queues from lengthening without limit.

(b) Once an eTPU thread starts to execute, it can execute for as long as it wishes in a single time slot. What are the major advantages and disadvantages of such an approach compared with the pre-emptive scheduling used by MicroC/OS?

Advantages: The eTPU time slot approach is very efficient because "unnecessary" context switches between tasks can be avoided. Each task can be structured to use time slots that are optimally sized to make good progress without overly hogging the eTPU scheduler.

Disadvantages: The eTPU threads/tasks/channels must be designed carefully to avoid hogging the scheduler. All possible worst-case combinations of threads must be considered to ensure that the system accomplishes all of its real-time objectives.